stable :

AndiEcker

May 14, 2024

CONTENTS

welcome to the documentation of the portions (app/service modules and sub-packages) of this freely extendable aedev namespace (**PEP 420**).

CHAPTER

ONE

FEATURES AND USE-CASES

the portions of this namespace are simplifying your Python application or service in the areas/domains:

- continuous integration
- continuous deployment
- outsourced text file (maintained centrally)
- code and config file templates
- git repository management (locally and remotely)
- software development
- software testing and QA

CHAPTER

TWO

CODE MAINTENANCE GUIDELINES

2.1 portions code requirements

- pure python
- fully typed (**PEP 526**)
- fully *documented*
- 100 % test coverage
- multi thread save
- code checks (using pylint and flake8)

2.2 design pattern and software principles

- DRY - don't repeat yourself
- KIS - keep it simple

2.3 contributing

we want to make it as easy and fun as possible for you to contribute to this project.

2.3.1 reporting bugs

before you create a new issue, please check to see if you are using the latest version of this project; the bug may already be resolved.

also search for similar problems in the issue tracker; it may already be an identified problem.

include as much information as possible into the issue description, at least:

1. version numbers (of Python and any involved packages).
2. small self-contained code example that reproduces the bug.
3. steps to reproduce the error.
4. any traceback/error/log messages shown.

2.3.2 requesting new features

1. on the git repository host server create new issue, providing a clear and detailed explanation of the feature you want and why it's important to add.
2. if you are able to implement the feature yourself (refer to the *contribution steps* section below).

2.3.3 contribution steps

thanks for your contribution – we'll get your merge request reviewed. you could also review other merge requests, just like other developers will review yours and comment on them. based on the comments, you should address them. once the reviewers approve, the maintainers will merge.

before you start make sure you have a GitLab account.

contribution can be done either with the *git-repo-manager tool* or directly by using the `git` command and the `Gitlab` server.

using the git repository manager *grm*

1. fork and clone the repository of this project to your computer

   in your console change the working directory to your project's parent folder. then run the following command with the `new_feature_or_fix` part replaced by an appropriate branch name, describing shortly your contribution:

   ```
   grm -b new_feature_or_fix fork aedev-group/aedev_aedev
   ```

   Note: the `grm fork` action will also add the forked repository as the remote `upstream` to your local repository.

   after the repository fork you change your current working directory to the new working tree root folder, created by the `grm fork` action, and execute the `grm renew` action. this will prepare a new package version of the project and upgrade the project files created from templates to its latest version.

2. code and check

   now use your favorite IDE/Editor to implement the new feature or code the bug fix. don't forget to amend the project with new unit and integrity tests, and ensure they pass, by executin from time to time the `grm check` action.

3. publish your changes

   before you initiate a push/merge request against the Gitlab server, execute the `grm prepare` action, which will create, with the help of the `git diff` command, a *.commit_msg.txt* file in the working tree root of your project, containing a short summary in the first line followed with a blank line and a list of the project files that got added, changed or deleted.

   Hint: the *.commit_msg.txt* file can be amended by any text editor before you run the `grm commit` action. for changes initiated by an issue please include the issue number (in the format `fixes #<issue-number>`) into this file. you may use `Markdown` syntax in this file for simple styling.

   to finally commit and upload your changes run the following three grm actions in the root folder of your project:

```
grm commit
grm push
grm request
```

the `grm commit` command is first executing a `grm check` action to do a finally check of the project resources and to run the unit and integrity tests. if all these checks pass then a new git commit will be created, including your changes to the project. `grm push``will then push the commit to your ``origin` remote repository (your fork) and `grm request` will finally create a bew merge/pull request against the `upstream` remote repository (the forked one).

Hint: to complete the workflow a maintainer of the project has to execute the `grm release` action. this will merge your changes into the main branch *develop* of the `upstream` repository and then release a new version of the project onto PyPI.

---

more detailed information of the features of the `grm` tool are available within the grm user manual.

using *git* and *Gitlab*

alternatively to the `grm` tool you could directly use the git command suite and the Gitlab website to achieve the same (with a lot more of typing and fiddling ;-):

1. fork the upstream repository into your user account.
2. clone your forked repo as `origin` remote to your computer, and add an `upstream` remote for the destination repo by running the following commands in the console of your local machine:

```
git clone https://gitlab.com/<YourGitLabUserName>/aedev_aedev.git
git remote add upstream https://gitlab.com/aedev-group/aedev_aedev.git
```

3. checkout out a new local feature branch and update it to the latest version of the `develop` branch:

```
git checkout -b <new_feature_or_fix_branch_name> develop
git pull --rebase upstream develop
```

   please keep your code clean by staying current with the `develop` branch, where code will be merged. if you find another bug, please fix it in a separated branch instead.

4. push the branch to your fork. treat it as a backup:

```
git push origin <new_feature_or_fix_branch_name>
```

5. code

   implement the new feature or the bug fix; include tests, and ensure they pass.

6. check

   run the basic code style and typing checks locally (pylint, mypy and flake8) before you commit.

7. commit

   for every commit please write a short summary in the first line followed with a blank line and then more detailed descriptions of the change. for bug fixes please include any issue number (in the format #nnn) in your summary:

```
git commit -m "issue #123: put change summary here (can be a issue title)"
```

> **Note:** **never leave the commit message blank!** provide a detailed, clear, and complete description of your changes!

8. publish your changes (prepare a Merge Request)

   before submitting a merge request, update your branch to the latest code:

   ```
   git pull --rebase upstream develop
   ```

   if you have made many commits, we ask you to squash them into atomic units of work. most issues should have one commit only, especially bug fixes, which makes them easier to back port:

   ```
   git checkout develop
   git pull --rebase upstream develop
   git checkout <new_feature_or_fix_branch_name>
   git rebase -i develop
   ```

   push changes to your fork:

   ```
   git push -f
   ```

9. issue/make a GitLab Merge Request:
   - navigate to your fork where you just pushed to
   - click *Merge Request*
   - in the branch field write your feature branch name (this is filled with your default branch name)
   - click *Update Commit Range*
   - ensure the changes you implemented are included in the *Commits* tab
   - ensure that the *Files Changed* tab incorporate all of your changes
   - fill in some details about your potential patch including a meaningful title
   - click *New merge request*.

2.3.4 release to PyPI

the release of a new/changed project will automatically be initiated by the GitLab CI, using the two protected vars `PYPI_USERNAME` and `PYPI_PASSWORD` (marked as masked) from the users group of this namespace, in order to provide the user name and password of the maintainers PyPI account (on Gitlab.com at Settings/CI_CD/Variables).

2.3.5 useful links and resources

- General GitLab documentation
- GitLab workflow documentation
- grm (git repository manager) module *`project repository`* and user manual

2.4 create new namespace

a **PEP 420** namespace splits the codebase of a library or framework into multiple project repositories, called portions (of the namespace).

Hint: the *aedev* namespace is providing the grm tool to create and maintain any namespace and its portions.

the id of a new namespace consists of letters only and has to be available on PYPI. the group-name name gets by default generated from the namespace name plus the suffix `'-group'`, so best choose an id that results in a group name that is available on your repository server.

2.5 register a new namespace portion

follow the steps underneath to add and register a new module as portion onto the *aedev* namespace:

1. open a console window and change the current directory to the parent directory of your projects root folders.
2. choose a not-existing/unique name for the new portion (referred as <*portion-name*> in the next steps).
3. run `grm --namespace=aedev --project=<portion_name> new-module` to register the portion name within the namespace, to create a new project folder *aedev_<portion-name>* (providing initial project files created from templates) and to get a pre-configured git repository (with the remote already set and the initial files unstaged, to be extended, staged and finally committed).
4. run `cd aedev_<portion-name>` to change the current to the working tree root of the new portion project.
5. run pyenv local venv_name (or any other similar tool) to create/prepare a local virtual environment.
6. fans of TDD are then coding unit tests in the prepared test module *test_aedev_<portion-name>.py*, situated within the *tests* sub-folder of your new code project folder.
7. extend the file <portion_name>.py situated in the *aedev* sub-folder to implement the new portion.
8. run `grm check-integrity` to run the linting and unit tests (if they fail go one or two steps back).
9. run `grm prepare`, then amend the commit message within the file *.commit_msg.txt*, then run `grm commit` and `grm push` to commit and upload your new portion to your personal remote/server repository fork, and finally run `grm request` to request the merge/pull into the forked/upstream repository in the users group *aedev-group* (at https://gitlab.com/aedev-group).

the registration of a new portion to the aedev namespace has to be done by a namespace maintainer.

registered portions will automatically be included into the *aedev namespace documentation*, available at ReadTheDocs.

CHAPTER

THREE

REGISTERED NAMESPACE PACKAGE PORTIONS

the following list contains all registered portions of the aedev namespace, plus additional modules of each portion.

Hint: a not on the ordering: portions with no dependencies are at the begin of the following list. the portions that are depending on other portions of the aedev namespace are listed more to the end.

aedev.setup_project	project setup helper functions
aedev.tpl_project	outsourced Python project files templates
aedev.tpl_namespace_root	templates and outsourced files for namespace root projects.
aedev.tpl_app	aedev_tpl_add module main module
aedev.git_repo_manager	create and maintain local/remote git repositories of Python projects
aedev.git_repo_manager.__main__	main module of git_repo_manager.
aedev.pythonanywhere	web api for www.pyanywhere.com and eu.pyanywhere.com
aedev.setup_hook	individually configurable setup hook

3.1 aedev.setup_project

3.1.1 project setup helper functions

this portion of the `aedev` namespace is providing constants and helper functions to install/setup Python projects of applications, modules, packages, namespace portions and their root packages via the `setuptools.setup()` function.

the function *project_env_vars()* is analyzing a Python project and is providing the project properties as a dict of project environment variable values.

the main goal of this project analysis is to:

1. ease the setup process of Python projects,
2. replace additional setup tools like e.g. *pipx* or *poetry* and
3. eliminate or at least minimize redundancies of the project properties, stored in the project files like `setup.py`, `setup.cfg`, `pyproject.toml`, ...

e.g. if you have to change the short description/title or the version number of a project you only need to edit them in one single place of your project. after that, the changed project property value will be automatically propagated/used in the next setup process.

basic helper functions

while *code_file_version()* determines the current version of any type of Python code file, *code_file_title()* does the same for the title of the code file's docstring.

package data resources of a project can be determined by calling the function *find_package_data()*. the return value can directly be passed to the *package_data* item of the kwargs passed to `setuptools.setup()`.

an optional namespace of a package gets determined and returned as string by the function *namespace_guess()*.

determine project environment variable values

the function *project_env_vars()* inspects the folder of a Python project to generate a complete mapping of environment variables representing project properties like e.g. names, ids, urls, file paths, versions or the content of the readme file.

if the current working directory is the root directory of a Python project to analyze then it can be called without specifying any arguments:

```
pev = project_env_vars()
```

to analyze a project in any other directory specify the path in the *project_path* argument:

```
pev = project_env_vars(project_path='path/to/project_or_parent')
```

..note::
if *project_env_vars()* gets called from within of your `setup.py` file, then a *True* value has to be passed to the keyword argument *from_setup*.

the project property values can be retrieved from the returned dictionary (the `pev` variable) either via the function *pev_str()* (only for string values), the function *pev_val()* or directly via getitem. the following example is retrieving a string reflecting the name of the project package:

```
project_name = pev_str(pev, 'project_name')
```

the type of project gets mapped by the *'project_type'* key. recognized project types are e.g. *a module*, *a package*, *a namespace root*, or an *gui application*.

if the current or specified directory to analyze is the parent directory of your projects (and is defined in *PARENT_FOLDERS*) then the mapped project type key will contain the special pseudo value *PARENT_PRJ*, which gets recognized by the *aedev.git_repo_manager* development tools e.g. for the creation of a new projects and the bulk processing of multiple projects or the portions of a namespace.

other useful properties in the *pev* mapping dictionary for real projects are e.g. *'project_version'* (determined e.g. from the __version__ module variable), *'repo_root'* (the url prefix to the remote/origin repositories host), or *'setup_kwargs'* (the keyword arguments passed to the `setuptools.setup()` function).

Hint: for a complete list of all available project environment variables check either the code of this module or the content of the returned *pev* variable (the latter can be done alternatively e.g. by running the *grm* tool with the command line options `-v`, `-D` and the command line argument `show-status`).

---

configure individual project environment variable values

the default values of project environment variables provided by this module are optimized for an easy maintenance of namespace portions like e.g. aedev and ae.

individual project environment variable values can be configured via the two files `pev.defaults` and `pev.updates`, which are situated in the working tree root folder of a project. the content of these files will be parsed by the built-in function `ast.literal_eval()` and has to result in a dict value. only the project environment variables that differ from the pev variable value have to be specified.

an existing `pev.defaults` is changing project environment variable default values which may affect other pev variables. e.g. to change the value of the author's name project environment variable *STK_AUTHOR*, the content of the file `pev.defaults` would look like:

```
{'STK_AUTHOR': "My Author Name"}
```

changing the default value of the *STK_AUTHOR* variable results that the variable `setup_kwargs['author']` will also have the value `"My Author Name"`.

in contrary the file `pev.updates` gets processed at the very end of the initialization of the project environment variables and the specified values get merged into the project environment variables with the help of the function `ae.base.deep_dict_update()`. therefore, putting the content from the last example into `pev.updates` will only affect the variable STK_AUTHOR, whereas `setup_kwargs['author']` will be left unchanged.

Hint: if code has to be executed to calculate an individual value of a project environment variable you have to modify the variable `pev` directly in your project's `setup.py` file. this can be achieved by either, providing a *setup-hook module* or by directly patching the `pev` variable, like shown in the following example:

```
from aedev.setup_project import project_env_vars

pev = project_env_vars(from_setup=True)
pev['STK_AUTHOR'] = "My Author Name"
...
```

Module Attributes

APP_PRJ	gui application project
DJANGO_PRJ	django website project
MODULE_PRJ	module portion/project
PACKAGE_PRJ	package portion/project
PARENT_PRJ	pseudo project type for new project started in parent-dir
ROOT_PRJ	namespace root project
NO_PRJ	no project detected
DOCS_HOST_PROTOCOL	documentation host connection protocol
DOCS_DOMAIN	documentation dns domain
DOCS_SUB_DOMAIN	doc sub domain; def: namespace|package+REPO_GROUP_SUFFIX
PARENT_FOLDERS	names of parent folders containing Python project directories
PYPI_PROJECT_ROOT	PYPI projects root url
MINIMUM_PYTHON_VERSION	minimum version of the Python/CPython runtime

continues on next page

Table 3.1 – continued from previous page

REPO_HOST_PROTOCOL	repo host connection protocol
REPO_CODE_DOMAIN	code repository dns domain (gitlab.com|github.com)
REPO_PAGES_DOMAIN	repository pages internet/dns domain
REPO_GROUP	repo users group name; def=namespace|package+REPO_GROUP_SUFFIX
REPO_GROUP_SUFFIX	repo users group name suffix (only used if REPO_GROUP is empty)
REQ_FILE_NAME	requirements default file name
REQ_DEV_FILE_NAME	default file name for development/template requirements
STK_AUTHOR	project author name default
STK_AUTHOR_EMAIL	project author email default
STK_LICENSE	project license default
STK_CLASSIFIERS	project classifiers defaults
STK_PYTHON_REQUIRES	default required Python version of project
VERSION_QUOTE	quote character of the __version__ number variable value
VERSION_PREFIX	search string to find the __version__ variable
DataFilesType	setup_kwargs['data_files']
PackageDataType	setup_kwargs['package_data']
SetupKwargsType	setuptools.setup()-kwargs
PevVarType	single project environment variable
PevType	project env vars mapping

Functions

code_file_title(file_name)	determine docstring title of a Python code file.
code_file_version(file_name)	read version of Python code file - from __version__ module variable initialization.
code_version(content)	read version of content string of a Python code file.
find_package_data(package_path)	find doc, template, kv, i18n translation text, image and sound files of an app or (namespace portion) package.
namespace_guess(project_path)	guess name of namespace name from the package/app/project root directory path.
pev_str(pev, var_name)	string value of project environment variable *var_name* of *pev*.
pev_val(pev, var_name)	determine value of project environment variable from passed pev or use module constant value as default.
project_env_vars([project_path, from_setup])	analyse and map the development environment of a package-/app-project into a dict of project property values.

APP_PRJ = 'app'

gui application project

DJANGO_PRJ = 'django'

django website project

MODULE_PRJ = 'module'

module portion/project

PACKAGE_PRJ = 'package'

package portion/project

PARENT_PRJ = 'projects-parent-dir'

pseudo project type for new project started in parent-dir

ROOT_PRJ = 'namespace-root'

namespace root project

NO_PRJ = ''

no project detected

DOCS_HOST_PROTOCOL = 'https://'

documentation host connection protocol

DOCS_DOMAIN = 'readthedocs.io'

documentation dns domain

DOCS_SUB_DOMAIN = ''

doc sub domain; def: namespace|package+REPO_GROUP_SUFFIX

PARENT_FOLDERS = ('Projects', 'PycharmProjects', 'esc', 'old_src', 'projects', 'repo', 'repos', 'source', 'src', 'docs')

names of parent folders containing Python project directories

PYPI_PROJECT_ROOT = 'https://pypi.org/project'

PYPI projects root url

MINIMUM_PYTHON_VERSION = '3.9'

minimum version of the Python/CPython runtime

REPO_HOST_PROTOCOL = 'https://'

repo host connection protocol

REPO_CODE_DOMAIN = 'gitlab.com'

code repository dns domain (gitlab.com|github.com)

REPO_PAGES_DOMAIN = 'gitlab.io'

repository pages internet/dns domain

REPO_GROUP = ''

repo users group name; def=namespace|package+REPO_GROUP_SUFFIX

REPO_GROUP_SUFFIX = '-group'

repo users group name suffix (only used if REPO_GROUP is empty)

REQ_FILE_NAME = 'requirements.txt'

requirements default file name

REQ_DEV_FILE_NAME = 'dev_requirements.txt'

default file name for development/template requirements

STK_AUTHOR = 'AndiEcker'

project author name default

STK_AUTHOR_EMAIL = 'aecker2@gmail.com'

project author email default

STK_LICENSE = 'OSI Approved :: GNU General Public License v3 or later (GPLv3+)'

project license default

STK_CLASSIFIERS = ['Development Status :: 3 - Alpha', 'License :: OSI Approved :: GNU General Public License v3 or later (GPLv3+)', 'Natural Language :: English', 'Operating System :: OS Independent', 'Programming Language :: Python', 'Programming Language :: Python :: 3', 'Programming Language :: Python :: 3.9', 'Topic :: Software Development :: Libraries :: Python Modules']

project classifiers defaults

STK_PYTHON_REQUIRES = '>=3.9'

default required Python version of project

VERSION_QUOTE = "'"

quote character of the __version__ number variable value

VERSION_PREFIX = "__version__ = '"

search string to find the __version__ variable

DataFilesType

setup_kwargs['data_files']

alias of `List[Tuple[str, Tuple[str, ...]]]`

PackageDataType

setup_kwargs['package_data']

alias of `Dict[str, List[str]]`

SetupKwargsType

setuptools.setup()-kwargs

alias of `Dict[str, Any]`

PevVarType

single project environment variable

alias of `str | Sequence[str] | List[Tuple[str, Tuple[str, ...]]] | Dict[str, Any]`

PevType

project env vars mapping

alias of `Dict[str, str | Sequence[str] | List[Tuple[str, Tuple[str, ...]]] | Dict[str, Any]]`

_compile_remote_vars(*pev*)

_compile_setup_kwargs(*pev*)

add setup kwargs from pev values, if not set in setup.cfg.

Parameters

pev (`Dict[str, Union[str, Sequence[str], List[Tuple[str, Tuple[str, ...]]], Dict[str, Any]]]`) – dict of project environment variables with a *'setup_kwargs'* dict to update/complete.

optional setup_kwargs for native/implicit namespace packages are e.g. *namespace_packages*. adding to setup_kwargs *include_package_data=True* results in NOT including package resources into sdist (if no MANIFEST.in file is used).

_init_defaults(*project_path*)

Return type
Dict[str, Union[str, Sequence[str], List[Tuple[str, Tuple[str, ...]]], Dict[str, Any]]]

_init_pev(*project_path*)

Return type
Dict[str, Union[str, Sequence[str], List[Tuple[str, Tuple[str, ...]]], Dict[str, Any]]]

_load_descriptions(*pev*)

load long description from the README file of the project.

Parameters
pev¶ (Dict[str, Union[str, Sequence[str], List[Tuple[str, Tuple[str, ...]]], Dict[str, Any]]]) – dict of project environment variables with a *'project_path'* key.

_load_requirements(*pev*)

load requirements from the available requirements.txt file(s) of this project.

Parameters
pev¶ (Dict[str, Union[str, Sequence[str], List[Tuple[str, Tuple[str, ...]]], Dict[str, Any]]]) – dict of project environment variables with the following required project env vars: DOCS_FOLDER, REQ_FILE_NAME, REQ_DEV_FILE_NAME, TESTS_FOLDER, namespace_name, project_name, project_path.

the project env vars overwritten in this argument by this function are: dev_require, docs_require, install_require, portions_packages, setup_require, tests_require.

code_file_title(*file_name*)

determine docstring title of a Python code file.

Parameters
file_name¶ (str) – name (and optional path) of module/script file to read the docstring title number from.

Return type
str

Returns
docstring title string or empty string if file|docstring-title not found.

code_file_version(*file_name*)

read version of Python code file - from __version__ module variable initialization.

Parameters
file_name¶ (str) – name (and optional path) of module/script file to read the version number from.

Return type
str

Returns
version number string or empty string if file or version-in-file not found.

code_version(*content*)

read version of content string of a Python code file.

Parameters
content¶ (Union[str, bytes]) – content of a code file, possibly containing the declaration of a *__version__* variable.

Return type
str

Returns
version number string or empty string if file or version-in-file not found.

find_package_data(*package_path*)

find doc, template, kv, i18n translation text, image and sound files of an app or (namespace portion) package.

Parameters
package_path¶ (str) – path of the root folder to collect from: project root for most projects or the package subdir (project_path/namespace_name(s)…/portion_name) for namespace projects.

Return type
Dict[str, List[str]]

Returns
setuptools package_data dict, where the key is an empty string (to be included for all sub-packages) and the dict item is a list of all found resource files with a relative path to the *package_path* directory. folder names with a leading underscore (like e.g. the docs *_build*, the PY_CACHE_FOLDER`|`__pycache__ and the *__enamlcache__* folders) get excluded. explicitly included will be any BUILD_CONFIG_FILE file, as well as any folder name starting with PACKAGE_INCLUDE_FILES_PREFIX (used e.g. by ae.updater), situated directly in the directory specified by *package_path*.

namespace_guess(*project_path*)

guess name of namespace name from the package/app/project root directory path.

Parameters
project_path¶ (str) – path to project root folder.

Return type
str

Returns
namespace import name of the project specified via the project root directory path.

pev_str(*pev*, *var_name*)

string value of project environment variable *var_name* of *pev*.

Parameters

- **pev**¶ (Dict[str, Union[str, Sequence[str], List[Tuple[str, Tuple[str, ...]]], Dict[str, Any]]]) – project environment variables dict.
- **var_name**¶ (str) – name of variable.

Return type
str

Returns
variable value or if not exists in pev then the constant/default value of this module or if there is no module constant with this name then an empty string.

Raises
AssertionError – if the specified variable value is not of type *str*. in this case use the function *pev_val()* instead.

Hint: the *str* type annotation of the return value makes mypy happy. additional the constant's values of this module will be taken into account. replaces *cast(str, pev.get('namespace_name', globals().get(var_name, ""))).*

pev_val(*pev*, *var_name*)

determine value of project environment variable from passed pev or use module constant value as default.

Parameters

- **pev** (Dict[str, Union[str, Sequence[str], List[Tuple[str, Tuple[str, ...]]], Dict[str, Any]]]) – project environment variables dict.
- **var_name** (str) – name of the variable to determine the value of.

Return type
Union[str, Sequence[str], List[Tuple[str, Tuple[str, ...]]], Dict[str, Any]]

Returns
project env var or module constant value. empty string if variable is not defined.

project_env_vars(*project_path=''*, *from_setup=False*)

analyse and map the development environment of a package-/app-project into a dict of project property values.

Parameters

- **project_path** (str) – optional rel/abs path of the package/app/project working tree root directory of a new or an existing project (default=current working directory).
- **from_setup** (bool) – pass True if this function get called from within the setup.py module of your project.

Return type
Dict[str, Union[str, Sequence[str], List[Tuple[str, Tuple[str, ...]]], Dict[str, Any]]]

Returns
dict/mapping with the determined project environment variable values.

3.2 aedev.tpl_project

3.2.1 outsourced Python project files templates

this package provides generic templates of basic project files for all types of Python projects. this includes project configuration files like e.g. *.gitignore*, as well as generic project documentation files like *CONTRIBUTING.rst* or *LICENSE.md*.

3.3 aedev.tpl_namespace_root

templates and outsourced files for namespace root projects.

3.4 aedev.tpl_app

aedev_tpl_add module main module

3.5 aedev.git_repo_manager

3.5.1 create and maintain local/remote git repositories of Python projects

this init module provides underneath some information on how to change/extend the code base of this tool the `grm` tool.

detailed information on the constants, functions and classes are available in *the module documentation*.

the installation and usage of this tool gets explained in a separate user manual document.

define a new action

to add a new action you only need to declare a new method or function decorated with the `_action()` decorator. the decorator will automatically register and integrate the new action into the `grm` tool.

file patching helper functions

this portion is also providing some helper functions to patch code and documentation files.

the functions *bump_file_version()*, *increment_version()*, *_git_project_version()* and *replace_file_version()* are incrementing any part of a version number of a module, portion, app or package.

templates are patched with the help of the functions *deploy_template()*, *patch_string()* and *refresh_templates()*.

in conjunction with the template projects of the *aedev* namespace (like e.g. *aedev.tpl_project*) any common portions file (even the `setup.py` file) can be created/maintained as a template in a single place, and then requested and updated individually for each portion project.

3.6 aedev.git_repo_manager.__main__

main module of git_repo_manager.

Module Attributes

ANY_PRJ_TYPE	tuple of available project types (including the pseudo-project-types: no-/incomplete-project and parent-folder)
ARG_MULTIPLES	mark multiple args in the *_action()* arg_names kwarg
ARG_ALL	*all* argument, used for lists e.g. of namespace portions.
ARGS_CHILDREN_DEFAULT	default arguments for children actions.
CMD_PIP	pip command using python venvs, especially on Windows
CMD_INSTALL	pip install command

continues on next page

Table 3.2 – continued from previous page

COMMIT_MSG_FILE_NAME	name of the file containing the commit message
DJANGO_EXCLUDED_FROM_CLEANUP	set of file path masks/pattern to exclude essential files from to be cleaned-up on the server.
GIT_FOLDER_NAME	git sub-folder in project path root of local repository
NULL_VERSION	initial package version number for new project
LOCK_EXT	additional file extension to block updates from templates
MAIN_BRANCH	main/develop/default branch name
MOVE_TPL_TO_PKG_PATH_NAME_PREFIX	template file/folder name prefix, to move the templates to the package path (instead of the project path); has to be specified after *SKIP_IF_PORTION_DST_NAME_PREFIX* (if both prefixes are needed).
OUTSOURCED_MARKER	to mark an outsourced project file, maintained externally
OUTSOURCED_FILE_NAME_PREFIX	file name prefix of outsourced/externally maintained file
PROJECT_VERSION_SEP	separates package name and version in pip req files
PPF(object)	
SKIP_IF_PORTION_DST_NAME_PREFIX	skip portion prj template dst root folder/file nam prefix
SKIP_PRJ_TYPE_FILE_NAME_PREFIX	file name prefix of skipped template if dst != prj type
TEMPLATE_PLACEHOLDER_ID_PREFIX	template id prefix marker
TEMPLATE_PLACEHOLDER_ID_SUFFIX	template id suffix marker
TEMPLATE_PLACEHOLDER_ARGS_SUFFIX	template args suffix marker
TEMPLATE_INCLUDE_FILE_PLACEHOLDER_ID	placeholder (func:*replace_with_file_content_or_default*)
TPL_FILE_NAME_PREFIX	file name prefix if template contains f-strings
TPL_IMPORT_NAME_PREFIX	package/import name prefix of template projects
TPL_STOP_CNV_PREFIX	file name prefix to support template of template
TPL_PACKAGES	import names of all possible template projects
TEMPLATES_FILE_NAME_PREFIXES	supported template file name prefixes (in the order they have to specified, apart from *TPL_STOP_CNV_PREFIX* which can be specified anywhere, to deploy template files to other template projects).
VERSION_MATCHER	pre-compiled regular expression to detect and bump the app/portion file version numbers of a version string.
ActionArgs	action arguments specified on grm command line
ActionFlags	action flags/kwargs specified on grm command line
RegisteredTemplateProject	registered template project info (tpl_projects item)
PdvType	project development variables type
ChildrenType	children pdv of a project parent or a namespace root
RepoType	repo host libs repo object (PyGithub, python-gitlab)
REGISTERED_ACTIONS	implemented actions registered via *_action()* deco
REGISTERED_HOSTS_CLASS_NAMES	class names of all supported remote host domains
REGISTERED_TPL_PROJECTS	projects providing templates and outsourced files
TEMP_CONTEXT	temp patch folder context (optional/lazy/late created)
cae	main app instance of this grm tool, initialized out of __name__ == '__main__' to be used for unit tests

Functions

activate_venv([name])	ensure to activate a virtual environment if it is different to the current one (the one on Python/app start).
active_venv()	determine the virtual environment that is currently active.
add_children_file(ini_pdv, file_name, ...)	add a file, template of outsourced text file to the working trees of parent/root and children/portions.
add_file(ini_pdv, file_name, rel_path)	add file, template or outsourced text file into the project working tree.
bump_file_version(file_name[, increment_part])	increment part of version number of module/script file, also removing any pre/alpha version sub-part/suffix.
bump_version(ini_pdv)	increment project version.
bytes_file_diff(file_content, file_path[, ...])	return the differences between the content of a file against a bytes array.
check_children_integrity(parent_pdv, ...)	run integrity checks for the specified children of a parent or portions of a namespace.
check_integrity(ini_pdv)	integrity check of files/folders completeness, outsourced/template files update-state and CI tests.
clone_children(parent_or_root_pdv, ...)	clone specified namespace-portion/parent-child repos to the local machine.
clone_project(ini_pdv[, package_or_portion])	clone remote repo to the local machine.
commit_children(ini_pdv, *children_pdv)	commit changes to children of a namespace/parent using the individually prepared commit message files.
commit_project(ini_pdv)	commit changes of a single project to the local repo using the prepared commit message file.
delete_children_file(ini_pdv, file_name, ...)	delete file or empty folder from parent/root and children/portions working trees.
delete_file(ini_pdv, file_name)	delete file or empty folder from project working tree.
deploy_template(tpl_file_path, dst_path, ...)	create/update outsourced project file content from a template.
editable_project_path(project_name)	determine the project path of a project package installed as editable.
find_extra_modules(project_path[, ...])	determine additional modules of a local (namespace portion) project.
find_git_branch_files(project_path[, ...])	find all added/changed/deleted/renamed/unstaged work-tree files that are not merged into the main branch.
find_project_files(project_path, root_path_masks)	find all files of a python package including the .py modules.
in_venv([name])	ensure the virtual environment gets activated within the context.
increment_version(version[, increment_part])	increment version number.
install_children_editable(ini_pdv, *children_pdv)	install parent children or namespace portions as editable on local machine.
install_editable(ini_pdv)	install project as editable from source/project root folder.
install_requirements(req_file[, project_path])	install requirements from requirements*.txt file with pip
main()	main app script
main_file_path(project_path, project_type, ...)	return the file path of the main/version type for the specified project type.
new_app(ini_pdv)	create or complete/renew a gui app project.
new_children(ini_pdv, *children_pdv)	initialize or renew parent folder children or namespace portions.

continues on next page

Table 3.3 – continued from previous page

new_django(ini_pdv)	create or complete/renew a django project.
new_module(ini_pdv)	create or complete/renew module project.
new_namespace_root(ini_pdv)	create or complete/renew namespace root package.
new_package(ini_pdv)	create or complete/renew package project.
new_project(ini_pdv)	complete/renew an existing project.
on_ci_host()	check and return True if this tool is running on the GitLab/GitHub CI host/server.
patch_string(content, pdv, **replacer)	replace f-string / dynamic placeholders in content with variable values / return values of replacer callables.
pdv_str(pdv, var_name)	string value of project development variable *var_name* of *pdv*.
pdv_val(pdv, var_name)	determine value of project development variable from passed pdv or `aedev_setup_project` module constant.
prepare_and_run_main()	prepare and run app
prepare_children_commit(ini_pdv, title, ...)	run code checks and prepare/overwrite the commit message file for a bulk-commit of children projects.
prepare_commit(ini_pdv[, title])	run code checks and prepare/overwrite the commit message file for the commit of a single project/package.
project_dev_vars([project_path])	analyse and map an extended project development environment, including template/root projects and git status.
project_version(imp_or_pkg_name, ...)	determine package name and version in list of package/version strings.
pypi_versions(pip_name)	determine all the available release versions of a package hosted at the PyPI 'Cheese Shop'.
refresh_children_outsourced(ini_pdv, ...)	refresh outsourced files from templates in namespace/project-parent children projects.
refresh_outsourced(ini_pdv)	refresh/renew all the outsourced files in the specified project.
refresh_templates(pdv[, logger])	convert ae namespace package templates found in the cwd or underneath (except excluded) to the final files.
rename_children_file(ini_pdv, old_file_name, ...)	rename file or folder in parent/root and children/portions working trees.
rename_file(ini_pdv, old_file_name, ...)	rename file or folder in project working tree.
replace_file_version(file_name[, ...])	replace version number of module/script file.
replace_with_file_content_or_default(args_str)	return file content if file name specified in first string arg exists, else return empty string or 2nd arg str.
root_packages_masks(pdv)	determine root sub packages from the passed project packages and add them glob path wildcards.
run_children_command(ini_pdv, command, ...)	run console command for the specified portions/children of a namespace/parent.
show_actions(ini_pdv)	get info of available/registered/implemented actions of the specified/current project and remote.
show_children_status(ini_pdv, *children_pdv)	run integrity checks for the specified portions/children of a namespace/parent.
show_children_versions(ini_pdv, *children_pdv)	show package versions (local, remote and on pypi) for the specified children of a namespace/parent.
show_status(ini_pdv)	show git status of the specified/current project and remote.
show_versions(ini_pdv)	display package versions of worktree, remote/origin repo, latest PyPI release and default app/web host.

continues on next page

Table 3.3 – continued from previous page

skip_files_lean_web(file_path)	file exclude callback to reduce the deployed files on the web server to the minimum.
skip_files_migrations(file_path)	file exclude callback for the files under the django migrations folders.
update_children(ini_pdv, *children_pdv)	fetch and rebase the MAIN_BRANCH to the local children repos of the parent/namespace-root(also updated).
update_project(ini_pdv)	fetch and rebase the MAIN_BRANCH of the specified project in the local repo.
venv_bin_path([name])	determine the absolute path of the bin/executables folder of a virtual pyenv environment.

Classes

GithubCom()	remote connection and actions on remote repo in gitHub.com.
GitlabCom()	remote connection and actions on gitlab.com.
PythonanywhereCom()	remote actions on remote web host pythonanywhere.com (to be specified by --domain option).
RemoteHost()	base class registering subclasses as remote host repo class in *REGISTERED_HOSTS_CLASS_NAMES*.

ANY_PRJ_TYPE = ('app', 'django', 'module', 'package', 'namespace-root')

tuple of available project types (including the pseudo-project-types: no-/incomplete-project and parent-folder)

ARG_MULTIPLES = ' ...'

mark multiple args in the *_action()* arg_names kwarg

ARG_ALL = 'all'

all argument, used for lists e.g. of namespace portions

ARGS_CHILDREN_DEFAULT = (('all',), ('children-sets-expr',), ('children-names ...',))

default arguments for children actions.

CMD_PIP = 'python -m pip'

pip command using python venvs, especially on Windows

CMD_INSTALL = 'python -m pip install'

pip install command

COMMIT_MSG_FILE_NAME = '.commit_msg.txt'

name of the file containing the commit message

DJANGO_EXCLUDED_FROM_CLEANUP = {'/django.mo', 'db.sqlite', 'media/**/*', 'project.db', 'static/**/*'}**

set of file path masks/pattern to exclude essential files from to be cleaned-up on the server.

GIT_FOLDER_NAME = '.git'

git sub-folder in project path root of local repository

NULL_VERSION = '0.0.0'

initial package version number for new project

LOCK_EXT = '.locked'

additional file extension to block updates from templates

MAIN_BRANCH = 'develop'

main/develop/default branch name

MOVE_TPL_TO_PKG_PATH_NAME_PREFIX = 'de_mtp_'

template file/folder name prefix, to move the templates to the package path (instead of the project path); has to be specified after *SKIP_IF_PORTION_DST_NAME_PREFIX* (if both prefixes are needed).

OUTSOURCED_MARKER = 'THIS FILE IS EXCLUSIVELY MAINTAINED'

to mark an outsourced project file, maintained externally

OUTSOURCED_FILE_NAME_PREFIX = 'de_otf_'

file name prefix of outsourced/externally maintained file

PROJECT_VERSION_SEP = '=='

separates package name and version in pip req files

PPF(*object*)

formatter for console printouts

SKIP_IF_PORTION_DST_NAME_PREFIX = 'de_sfp_'

skip portion prj template dst root folder/file nam prefix

SKIP_PRJ_TYPE_FILE_NAME_PREFIX = 'de_spt_'

file name prefix of skipped template if dst != prj type

TEMPLATE_PLACEHOLDER_ID_PREFIX = '# '

template id prefix marker

TEMPLATE_PLACEHOLDER_ID_SUFFIX = '#('

template id suffix marker

TEMPLATE_PLACEHOLDER_ARGS_SUFFIX = ')#'

template args suffix marker

TEMPLATE_INCLUDE_FILE_PLACEHOLDER_ID = 'IncludeFile'

placeholder (func:*replace_with_file_content_or_default*)

TPL_FILE_NAME_PREFIX = 'de_tpl_'

file name prefix if template contains f-strings

TPL_IMPORT_NAME_PREFIX = 'aedev.tpl_'

package/import name prefix of template projects

TPL_STOP_CNV_PREFIX = '_z_'

file name prefix to support template of template

TPL_PACKAGES = ['aedev.tpl_app', 'aedev.tpl_django', 'aedev.tpl_module', 'aedev.tpl_package', 'aedev.tpl_namespace_root', 'aedev.tpl_project']

import names of all possible template projects

TEMPLATES_FILE_NAME_PREFIXES = ('de_sfp_', 'de_spt_', 'de_otf_', 'de_tpl_', '_z_')

supported template file name prefixes (in the order they have to specified, apart from *TPL_STOP_CNV_PREFIX* which can be specified anywhere, to deploy template files to other template projects).

Hint: *SKIP_IF_PORTION_DST_NAME_PREFIX* can also be path name prefix, like *MOVE_TPL_TO_PKG_PATH_NAME_PREFIX*.

VERSION_MATCHER = re.compile("^__version__ = '(\\d+)[.](\\d+)[.](\\d+)[a-z\\d]*'", re.MULTILINE)

pre-compiled regular expression to detect and bump the app/portion file version numbers of a version string.

The version number format has to be **conform to PEP396** and the sub-part to Pythons distutils (trailing version information indicating sub-releases, are either "a1,a2,...,aN" (for alpha releases), "b1,b2,...,bN" (for beta releases) or "pr1,pr2,...,prN" (for pre-releases). Note that distutils got deprecated in Python 3.12 (see package `packaging.version` as replacement)).

ActionArgs

action arguments specified on grm command line

alias of `List[str]`

ActionFlags

action flags/kwargs specified on grm command line

alias of `Dict[str, Any]`

RegisteredTemplateProject

registered template project info (tpl_projects item)

alias of `Dict[str, str]`

PdvType

project development variables type

alias of `Dict[str, Any]`

ChildrenType

children pdv of a project parent or a namespace root

alias of `OrderedDict[str, Dict[str, Any]]`

RepoType

repo host libs repo object (PyGithub, python-gitlab)

alias of `Repository | Project`

REGISTERED_ACTIONS: Dict[str, Dict[str, Any]] = {'GithubCom.fork_project': {'annotations': {'forked_usr_repo': <class 'str'>, 'ini_pdv': typing.Dict[str, typing.Any]}, 'arg_names': (('forked-user-slash-repo',),), 'docstring': ' create/renew fork of a remote repo specified via the 1st argument, into our user/group namespace. ', 'local_action': False, 'project_types': ('projects-parent-dir', 'app', 'django', 'module', 'package', 'namespace-root'), 'shortcut': 'fork'}, 'GithubCom.push_project': {'annotations': {'branch_name': <class 'str'>, 'ini_pdv': typing.Dict[str, typing.Any]}, 'arg_names': ((), ('branch-name',)), 'docstring': ' push current/specified branch of project/package to remote host domain, version-tagged if release is True.\n\n :param branch_name: optional branch name to push (alternatively specified by the ``branch`` command line\n option).\n ', 'local_action': False, 'project_types': ('app', 'django', 'module', 'package', 'namespace-root'), 'shortcut': 'push'}, 'GithubCom.release_project': {'annotations': {'ini_pdv': typing.Dict[str, typing.Any], 'version_tag': <class 'str'>}, 'arg_names': (('version-tag',), ('LATEST',)), 'docstring': ' update local MAIN_BRANCH from origin and if pip_name is set then also release the latest/specified version.\n\n :param version_tag: push version tag in the format ``v<version-number>`` to release or ``LATEST`` to use\n the version tag of the latest git repository version.\n ', 'local_action': False, 'project_types': ('app', 'django', 'module', 'package', 'namespace-root'), 'shortcut': 'release'}, 'GithubCom.request_merge': {'annotations': {'ini_pdv': typing.Dict[str, typing.Any]}, 'docstring': ' request merge of the origin=fork repository into the main branch at remote/upstream=forked. ', 'local_action': False, 'project_types': ('app', 'django', 'module', 'package', 'namespace-root'), 'shortcut': 'request'}, 'GitlabCom.clean_releases': {'annotations': {'ini_pdv': typing.Dict[str, typing.Any], 'return': typing.List[str]}, 'docstring': ' delete local+remote release tags and branches of the specified project that got not published to PYPI. ', 'local_action': False, 'project_types': ('app', 'django', 'module', 'package', 'namespace-root')}, 'GitlabCom.fork_children': {'annotations': {'children_pdv': typing.Dict[str, typing.Any], 'ini_pdv': typing.Dict[str, typing.Any]}, 'arg_names': (('all',), ('children-sets-expr',), ('children-names ...',)), 'docstring': ' fork children of a namespace root project or of a parent folder. ', 'local_action': False, 'project_types': ('projects-parent-dir', 'namespace-root')}, 'GitlabCom.fork_project': {'annotations': {'ini_pdv': typing.Dict[str, typing.Any]}, 'docstring': ' create/renew fork of a remote repo specified via the ``package`` option, into our user/group namespace. ', 'local_action': False, 'project_types': ('projects-parent-dir', 'app', 'django', 'module', 'package', 'namespace-root'), 'shortcut': 'fork'}, 'GitlabCom.push_children': {'annotations': {'children_pdv': typing.Dict[str, typing.Any], 'ini_pdv': typing.Dict[str, typing.Any]}, 'arg_names': (('all',), ('children-sets-expr',), ('children-names ...',)), 'docstring': ' push specified children projects to remotes/origin. ', 'local_action': False, 'project_types': ('projects-parent-dir', 'namespace-root'), 'shortcut': 'push'}, 'GitlabCom.push_project': {'annotations': {'branch_name': <class 'str'>, 'ini_pdv': typing.Dict[str, typing.Any]}, 'arg_names': ((), ('branch-name',)), 'docstring': ' push current/specified branch of project/package to remote host domain, version-tagged if release is True.\n\n :param branch_name: optional branch name to push (alternatively specified by the ``branch`` command line\n option).\n ', 'local_action': False, 'project_types': ('app', 'django', 'module', 'package', 'namespace-root'), 'shortcut': 'push'}, 'GitlabCom.release_children': {'annotations': {'children_pdv': typing.Dict[str, typing.Any], 'ini_pdv': typing.Dict[str, typing.Any]}, 'arg_names': (('all',), ('children-sets-expr',), ('children-names ...',)), 'docstring': ' release the latest versions of the specified parent/root children projects to remotes/origin. ', 'local_action': False, 'project_types': ('projects-parent-dir', 'namespace-root'), 'shortcut': 'release'}, 'GitlabCom.release_project': {'annotations': {'ini_pdv': typing.Dict[str, typing.Any], 'version_tag': <class 'str'>}, 'arg_names': (('version-tag',), ('LATEST',)), 'docstring': ' update local MAIN_BRANCH from origin and if pip_name is set then also release the latest/specified version.\n\n :param version_tag: push version tag in the format ``v<version-number>`` to release or ``LATEST`` to use\n the version tag of the latest git repository version.\n ', 'local_action': False, 'project_types': ('app', 'django', 'module', 'package', 'namespace-root'), 'shortcut': 'release'}, 'GitlabCom.request_children_merge': {'annotations': {'children_pdv': typing.Dict[str,

implemented actions registered via *_action()* deco

_RCS: Dict[str, Callable] = {'_git_add': <function _git_add>, 'module.makedirs': <function makedirs>, 'module.write_file': <function write_file>}

registered recordable callees, for check* actions, using other actions with temporary redirected callees.

REGISTERED_HOSTS_CLASS_NAMES: Dict[str, str] = {'github.com': 'GithubCom', 'gitlab.com': 'GitlabCom', 'pythonanywhere.com': 'PythonanywhereCom'}

class names of all supported remote host domains

REGISTERED_TPL_PROJECTS: Dict[str, Dict[str, str]] = {}

projects providing templates and outsourced files

TEMP_CONTEXT: TemporaryDirectory | None = None

temp patch folder context (optional/lazy/late created)

TEMP_PARENT_FOLDER: str

temporary parent folder for to clone git repos into

cae = <ae.console.ConsoleApp object>

main app instance of this grm tool, initialized out of __name__ == '__main__' to be used for unit tests

_action(**project_types*, ***deco_kwargs*)

parametrized decorator to declare functions and *RemoteHost* methods as *grm* actions.

Return type
Callable

_recordable_function(*callee*)

decorator to register function as recordable (to be replaced/redirected in protocol mode).

Return type
Callable

_rc_id(*instance*, *method_name*)

compile recordable callee id of object method or module instance attribute/function.

Return type
str

_record_calls(**recordable_methods*, ***recordable_functions*)

Return type
Iterator[None]

activate_venv(*name=''*)

ensure to activate a virtual environment if it is different to the current one (the one on Python/app start).

Parameters
name¶ (str) – the name of the venv to activate. if this arg is empty or not specified then the venv of the project in the current working directory tree will be activated.

Return type
str

Returns
the name of the previously active venv or an empty string if the requested or no venv was active, or if venv is not supported.

active_venv()

determine the virtual environment that is currently active.

Note: the current venv gets set via *data: `os.environ* on start of this Python app or by *activate_venv()*.

Return type
str

Returns
the name of the currently active venv.

bump_file_version(*file_name*, *increment_part=3*)

increment part of version number of module/script file, also removing any pre/alpha version sub-part/suffix.

Parameters

- **file_name** (str) – module/script file name to be patched/version-bumped.
- **increment_part** (int) – version number part to increment: 1=mayor, 2=minor, 3=build/revision (default=3).

Return type
str

Returns
empty string on success, else error string.

bytes_file_diff(*file_content*, *file_path*, *line_sep='\\n'*)

return the differences between the content of a file against a bytes array.

Parameters

- **file_content** (bytes) – older file bytes to be compared against the file content of the file specified by the *file_path* argument.
- **file_path** (str) – path to the file of which newer content gets compared against the file bytes specified by the *file_content* argument.
- **line_sep** (str) – string used to prefix, separate and indent the lines in the returned output string.

Return type
str

Returns
differences between the two file contents, compiled with the *git diff* command.

deploy_template(*tpl_file_path*, *dst_path*, *patcher*, *pdv*, *logger=<built-in function print>*, *replacer=None*, *dst_files=None*)

create/update outsourced project file content from a template.

Parameters

- **tpl_file_path** (str) – template file path/name.ext (absolute or relative to current working directory).
- **dst_path** (str) – absolute or relative destination path without the destination file name. relative paths are relative to the project root path (the *project_path* item in the *pdv* argument).

- **patcher** (str) – patching template project or function (to be added into the outsourced project file).
- **pdv** (Dict[str, Any]) – project env/dev variables dict of the destination project to patch/refresh, providing values for (1) f-string template replacements, and (2) to specify the project type, and root or package data folder (in the *project_type*, and *project_path* or *package_path* items).
- **logger** (Callable) – print()-like callable for logging.
- **replacer** (Optional[Dict[str, Callable[[str], str]]]) – optional dict with multiple replacer: key=placeholder-id and value=replacer callable.
- **dst_files** (Optional[Set[str]]) – optional set of project file paths to be excluded from to be created/updated. if the project file got created/updated by this function then the destination file path will be added to this set.

Return type
bool

Returns
True if template got deployed/written to the destination, else False.

Note: the project file will be kept unchanged if either:

- the absolute file path is in the set specified by the *dst_files* argument,
- there exists a lock-file with the additional *LOCK_EXT* file extension, or
- the outsourced project text does not contain the *OUTSOURCED_MARKER* string.

---

editable_project_path(*project_name*)

determine the project path of a project package installed as editable.

Parameters
project_name (str) – project package name to search for.

Return type
str

Returns
project source root path of an editable installed package or empty string if not found as editable installed package.

find_extra_modules(*project_path*, *namespace_name=''*, *portion_name=''*)

determine additional modules of a local (namespace portion) project.

Parameters

- **project_path** (str) – file path of the local namespace project root directory/folder. passing an empty string will search in the current working directory.
- **namespace_name** (str) – namespace name or pass an empty string for non-namespace-portion projects.
- **portion_name** (str) – name of the portion (folder). pass an empty string for non-namespace-portion projects.

Return type
List[str]

Returns
list of module import name strings (without file extension and path separators as dots). modules in the `TEMPLATES_FOLDER` and any `PY_INIT` modules are excluded.

find_git_branch_files(*project_path*, *branch_or_tag='develop'*, *untracked=False*, *skip_file_path=<function <lambda>>*)

find all added/changed/deleted/renamed/unstaged worktree files that are not merged into the main branch.

Parameters

- **project_path** (str) – path of the project root folder. pass empty string to use the current working directory.
- **branch_or_tag** (str) – branch(es)/tag(s)/commit(s) passed to git diff to specify the changed files between version(s).
- **skip_file_path** (Callable[[str], bool]) – called for each found file passing the file path relative to the project root folder (specified by the *project_path* argument), returning `True` to exclude/skip the file with passed file path.
- **untracked** (bool) – pass True to include untracked files from the returned result set.

Return type
Set[str]

Returns
set of file paths relative to worktree root specified by the project root path specified by the *project_path* argument.

find_project_files(*project_path*, *root_path_masks*, *skip_file_path=<function <lambda>>*)

find all files of a python package including the .py modules.

Parameters

- **project_path** (str) – path of the project root folder. pass empty string to use the current working directory.
- **root_path_masks** (List[str]) – list of folder or sub-package path masks with wildcards, relative to the project root.
- **skip_file_path** (Callable[[str], bool]) – called for each found file with their file path (relative to project root folder in *project_path*) as argument, returning `True` to exclude/skip the specified file.

Return type
set[str]

Returns
set of file paths relative to the project root folder specified by the argument *project_path*.

increment_version(*version*, *increment_part=3*)

increment version number.

Parameters

- **version** (Union[str, Iterable[str]]) – version number string or an iterable of version string parts.
- **increment_part** (int) – part of the version number to increment (1=mayor, 2=minor, 3=patch).

Return type
str

Returns
incremented version number.

`install_requirements`(*req_file*, *project_path=''*)

install requirements from requirements*.txt file with pip

Parameters

- **req_file** (str) – pip requirements.txt file path.
- **project_path** (str) – project root folder path.

Returns
0/zero on installation without errors, else pip error return code.

`in_venv`(*name=''*)

ensure the virtual environment gets activated within the context.

Parameters
name (str) – the name of the venv to activate. if not specified then the venv of the project in the current working directory tree will be activated.

Return type
Iterator[None]

`main_file_path`(*project_path*, *project_type*, *namespace_name*)

return the file path of the main/version type for the specified project type.

Parameters

- **project_path** (str) – project path, including the package name as basename.
- **project_type** (str) – project type to determine the main/version file path for.
- **namespace_name** (str) – namespace name if for namespace portion or root projects, else pass empty string.

Return type
str

Returns
main file path and name.

`on_ci_host`()

check and return True if this tool is running on the GitLab/GitHub CI host/server.

Return type
bool

Returns
True if running on CI host, else False

`project_version`(*imp_or_pkg_name*, *packages_versions*)

determine package name and version in list of package/version strings.

Parameters

- **imp_or_pkg_name** (str) – import or package name to search.
- **packages_versions** (List[str]) – project package versions string: <project_name>[<PROJECT_VERSION_SEP><project_version>].

Return type
Sequence[str]

Returns
sequence of package name and version number. the package name is an empty string if it is not in *packages_versions*. the version number is an empty string if no package version is specified in *packages_versions*.

patch_string(*content*, *pdv*, ***replacer*)

replace f-string / dynamic placeholders in content with variable values / return values of replacer callables.

Parameters

- **content** (str) – f-string to patch (e.g. a template file's content).
- **pdv** (Dict[str, Any]) – project env/dev vars dict with variables used as globals for f-string replacements.
- **replacer** (Callable[[str], str]) – optional kwargs dict with key/name=placeholder-id and value=replacer-callable. if not passed then the replacer with id TEMPLATE_INCLUDE_FILE_PLACEHOLDER_ID will be searched and if found the callable *replace_with_file_content_or_default()* will be executed.

Return type
str

Returns
string extended with include snippets found in the same directory.

Raises
Exception – if evaluation of :paramref;`~patch_string.content` f-string failed (because of missing-globals-NameError/SyntaxError/ValueError/...).

pdv_str(*pdv*, *var_name*)

string value of project development variable *var_name* of *pdv*.

Parameters

- **pdv** (Dict[str, Any]) – project development variables dict.
- **var_name** (str) – name of variable.

Return type
str

Returns
variable value or if not exists in pdv then the constant/default value of the module `aedev_setup_project` or if no constant with this name exists then an empty string.

Raises
AssertionError – if the specified variable value is not of type *str*. in this case use the function *pdv_val()* instead.

pdv_val(*pdv*, *var_name*)

determine value of project development variable from passed pdv or `aedev_setup_project` module constant.

Parameters

- **pdv** (Dict[str, Any]) – project environment variables dict.
- **var_name** (str) – name of the variable to determine the value of.

Return type
Any

Returns
project env var or module constant value. empty string if variable is not defined.

project_dev_vars(*project_path=''*)

analyse and map an extended project development environment, including template/root projects and git status.

Parameters
project_path¶ (str) – optional rel/abs path of the package/app/project root directory of a new and existing project (defaults to the current working directory if empty or not passed).

Return type
Dict[str, Any]

Returns
dict/mapping with the determined project development variable values.

pypi_versions(*pip_name*)

determine all the available release versions of a package hosted at the PyPI 'Cheese Shop'.

Parameters
pip_name¶ (str) – pip/package name to get release versions from.

Return type
List[str]

Returns
list of released versions (the latest last) or on error a list with a single empty string item.

refresh_templates(*pdv*, *logger=<built-in function print>*, ***replacer*)

convert ae namespace package templates found in the cwd or underneath (except excluded) to the final files.

Parameters

- **pdv**¶ (Dict[str, Any]) – project env/dev variables dict of the destination project to patch/refresh, providing values for (1) f-string template replacements, and (2) to control the template registering, patching and deployment via the variables:
  - *namespace_name*: namespace of the destination project.
  - *package_path*: path to package data root of the destination project.
  - *project_path*: path to working tree root of the destination project.
  - 'project_name': pypi name of the package/portion/app/.. project.
  - *project_type*: type of the destination project.
  - *repo_url*: remote/upstream repository url of the destination project.
  - *tpl_projects*: template projects data (import name, project path and version).

  Hint: use the function *project_dev_vars()* to create this dict.

- **logger**¶ (Callable) – print()-like callable for logging.
- **replacer**¶ (Callable[[str], str]) – dict of optional replacer with key=placeholder-id and value=callable. if not passed then only the replacer with id TEMPLATE_INCLUDE_FILE_PLACEHOLDER_ID and its callable/func *replace_with_file_content_or_default()* will be executed.

Return type
Set[str]

Returns
set of patched destination file names.

replace_file_version(*file_name*, *increment_part=0*, *new_version=''*)

replace version number of module/script file.

Parameters

- **file_name** (str) – module/script file name to be patched/version-bumped.
- **increment_part** (int) – version number part to increment: 1=mayor, 2=minor, 3=build/revision, default 0=nothing.
- **new_version** (str) – if passed replaces the original version in the file.

Return type
str

Returns
empty string on success, else error string.

replace_with_file_content_or_default(*args_str*)

return file content if file name specified in first string arg exists, else return empty string or 2nd arg str.

Parameters
args_str (str) – pass either file name, or file name and default literal separated by a comma character. spaces, tabs and newline characters get removed from the start and end of the file name. a default literal gets parsed like a config variable, the literal value gets return.

Return type
str

Returns
file content or default literal value or empty string (if file not exists and there is no comma character in *args_str*).

root_packages_masks(*pdv*)

determine root sub packages from the passed project packages and add them glob path wildcards.

Parameters
pdv (Dict[str, Any]) – project environment variables dict.

Return type
list[str]

Returns
list of project root packages extended with glob path wildcards.

skip_files_migrations(*file_path*)

file exclude callback for the files under the django migrations folders.

Parameters
file_path (str) – path to file to check for exclusion, relative to the project root folder.

Return type
bool

Returns
True if the file specified in *file_path* has to excluded, else False.

skip_files_lean_web(*file_path*)

file exclude callback to reduce the deployed files on the web server to the minimum.

Parameters
file_path (str) – path to file to check for exclusion, relative to the project root folder.

Return type
bool

Returns
True if the file specified in *file_path* has to excluded, else False.

venv_bin_path(*name=''*)

determine the absolute path of the bin/executables folder of a virtual pyenv environment.

Parameters
name¶ (str) – the name of the venv. if not specified then the venv name will be determined from the first found `.python-version` file, starting in the current working directory (cwd) and up to 3 parent directories above.

Return type
str

Returns
absolute path of the bin folder of the projects local pyenv virtual environment

_act_callable(*ini_pdv*, *act_name*)

Return type
Optional[Callable]

_act_spec(*pdv*, *act_name*)

Return type
Tuple[Dict[str, Any], str]

_available_actions(*project_type=<ae.base.UnsetType object>*)

Return type
Set[str]

_chk_if(*error_code*, *check_result*, *error_message*)

exit/quit this console app if the *check_result* argument is False and the *force* app option is False.

_check_commit_msg_file(*pdv*)

Return type
str

_check_folders_files_completeness(*pdv*)

_check_children_not_exist(*parent_or_root_pdv*, **project_versions*)

_check_resources_img(*pdv*)

check images, message texts and sounds of the specified project.

Return type
List[str]

_check_resources_i18n_ae(*file_name*, *content*)

check a translation text file with ae_i18n portion message texts.

Parameters

- **file_name**¶ (str) – message texts file name.
- **content**¶ (str) – message texts file content.

_check_resources_i18n_po(*file_name*, *content*)

check a translation text file with GNU gettext message texts.

Parameters

- **file_name** (str) – message texts file name (.po file).
- **content** (str) – message texts file content.

_check_resources_i18n_texts(*pdv*)

Return type
List[str]

_check_resources_snd(*pdv*)

Return type
List[str]

_check_resources(*pdv*)

check images, message texts and sounds of the specified project.

_check_templates(*pdv*)

_check_types_linting_tests(*pdv*)

_children_desc(*pdv*, *children_pdv=()*)

Return type
str

_children_project_names(*ini_pdv*, *names*, *chi_vars*)

Return type
List[str]

_children_path_package_option_reset()

_cl(*err_code*, *command_line*, *extra_args=()*, *lines_output=None*, *exit_on_err=True*, *exit_msg=''*, *shell=False*)

execute command in the current working directory of the OS console/shell, dump error and exit app if needed.

Parameters

- **err_code** (int) – error code to pass to console as exit code (if *exit_on_err* is True).
- **command_line** (str) – command line string to execute on the console/shell. could contain command line args separated by whitespace characters (alternatively use `extra_args`).
- **extra_args** (Sequence) – optional sequence of extra command line arguments.
- **lines_output** (Optional[List[str]]) – optional list to return the lines printed to stdout/stderr on execution.
- **exit_on_err** (bool) – pass False to **not** exit the app on error (*exit_msg* has then to be empty).
- **exit_msg** (str) – additional text to print on stdout/console if error and *exit_on_err* is True.
- **shell** (bool) – pass True to execute command in the default OS shell (see `subprocess.run()`).

Return type
int

Returns
0 on success or the error number if an error occurred.

_clone_template_project(*import_name*, *version*)

Return type
str

_debug_or_verbose()
determine if verbose or debug option got specified (preventing on app init early call of cae.get_option()).

Return type
bool

_exit_error(*error_code*, *error_message=''*)
quit this shell script, optionally displaying an error message.

_expected_args(*act_spec*)

Return type
str

_get_branch(*pdv*)

Return type
str

_get_host_class_name(*host_domain*)

Return type
str

_get_host_config_val(*host_domain*, *option_name*, *host_user=''*, *name_prefix='repo'*)
determine host domain, group, user and credential values.

Parameters

- **host_domain** (str) – domain name of the host. pass empty string to skip search for host-specific variable.
- **option_name** (str) – host option name and config variable name part ('domain', 'group', 'user', 'token'), resulting in e.g. user, repo_user, repo_user_at_xxx, web_user, web_user_at…
- **host_user** (str) – username at the host. if not passed or *host_domain* is empty then skip the search for a user-specific variable value.
- **name_prefix** (str) – config variable name prefix. pass 'web' to get web server host config values.

Return type
Optional[str]

Returns
config variable value or None if not found.

_get_host_domain(*pdv*, *name_prefix='repo'*)
determine domain name of repository|web host from –domain option, repo_domain or web_domain config variable.

Parameters
name_prefix (str) – config variable name prefix. pass 'web' to get web server host config values.

Return type
str

Returns
domain name of repository|web host.

_get_host_group(*pdv*, *host_domain*)

determine user group name from –group option or repo_group config variable.

Parameters
host_domain (str) – domain to get user token for.

Return type
str

Returns
user group name or if not found the default username STK_AUTHOR.

_get_host_user_name(*pdv*, *host_domain*, *name_prefix='repo'*)

determine username from –user option, repo_user or web_user config variable.

Parameters

- **host_domain** (str) – domain to get user token for.
- **name_prefix** (str) – config variable name prefix. pass 'web' to get web server host config values.

Return type
str

Returns
username or if not found the user group name.

_get_host_user_token(*host_domain*, *host_user=''*, *name_prefix='repo'*)

determine token or password of user from –token option, repo_token or web_token config variable.

Parameters

- **host_domain** (str) – domain to get user token for.
- **host_user** (str) – host user to get token for.
- **name_prefix** (str) – config variable name prefix. pass 'web' to get web server host config values.

Return type
str

Returns
token string for domain and user on repository|web host.

_get_namespace(*pdv*, *project_type*)

Return type
str

_get_parent_path(*pdv*)

Return type
str

_get_parent_packageversion(*pdv*, *package_or_portion*)

Return type
Tuple[str, str]

_get_path_package(*pdv*, *project_type=''*)

Return type
Tuple[str, str, str]

_get_prj_name(*pdv*, *project_type=''*)

Return type
str

_get_renamed_path_package(*pdv*, *namespace_name*, *project_type*)

Return type
Tuple[str, str]

_git_add(*pdv*)

_git_branches(*pdv*)

Return type
List[str]

_git_checkout(*pdv*, **extra_args*, *branch=''*, *from_branch=''*)

_git_clone(*repo_root*, *project_name*, *branch_or_tag=''*, *parent_path=''*, *extra_args=()*)

Return type
str

_git_commit(*pdv*, *extra_options=()*)

execute the command 'git commit' for the specified project.

Parameters

- **pdv** (Dict[str, Any]) – providing project-name and -path in which this git command gets executed.
- **extra_options** (Iterable[str]) – additional options passed to *git commit* command line, e.g. ["–patch", "–dry-run"].

Note: ensure the commit message in the file *COMMIT_MSG_FILE_NAME* is uptodate.

_git_current_branch(*pdv*)

Return type
str

_git_diff(*pdv*, **extra_opt_and_ref_specs*)

Return type
List[str]

_git_fetch(*pdv*, **extra_args*)

Return type
List[str]

_git_init_if_needed(*pdv*)

Return type
bool

_git_merge(*pdv*, *from_branch*)

Return type
bool

_git_project_version(*pdv*, *increment_part=3*)

determine latest or the next free package git repository version or the project specified via the pdv argument.

Parameters

- **pdv** (Dict[str, Any]) – project dev vars to identify the package.
- **increment_part** (int) – part of the version number to be incremented (1=mayor, 2=minor/namespace, 3=patch). pass zero/0 to return the latest published package version.

Return type
str

Returns
latest published repository package version as string or the first version (increment_version(NULL_VERSION, increment_part) or "0.0.1") if project never published a version tag to remotes/origin or an empty string on error.

_git_push(*pdv*, **branches_and_tags*, *exit_on_error=True*, *extra_args=()*)

push portion in the current working directory to the specified branch.

Return type
int

_git_remotes(*pdv*)

Return type
Dict[str, str]

_git_renew_remotes(*pdv*)

_git_status(*pdv*)

Return type
List[str]

_git_tag_add(*pdv*, *tag*)

_git_tag_in_branch(*pdv*, *tag*, *branch='origin/develop'*)

check if tag/ref is in the specified or in the remote origin main branch.

Parameters

- **pdv** (Dict[str, Any]) – project vars.
- **tag** (str) – any ref like a tag or another branch, to be searched within *branch*.
- **branch** (str) – branch to be searched in for *tag*.

Return type
bool

Returns
`True` if ref got found in branch, else `False`.

_git_tag_list(*pdv*, *tag_pattern='v*'*)

Return type
List[str]

_git_uncommitted(*pdv*)

Return type
List[str]

_hint(*act_fun*, *run_grm_message_suffix=''*)

Return type
str

_in_prj_dir_venv(*project_path*, *venv_name=''*)

Return type
Iterator[None]

_init_act_args_check(*ini_pdv*, *act_spec*, *act_name*, *act_args*, *act_flags*)

check and possibly complete the command line arguments, and split optional action flags from action args.

called after _init_act_exec_args/INI_PDV-initialization.

_init_act_args_shortcut(*ini_pdv*, *ini_act_name*)

Return type
str

_init_act_exec_args()

prepare execution of action requested via command line arguments and options.

- init project dev vars
- check if action is implemented
- check action arguments
- run optional pre_action.

Return type
Tuple[Dict[str, Any], str, tuple, Dict[str, Any]]

Returns
tuple of project pdv, action name to execute, a tuple with additional action arguments, and a dict of optional action flag arguments.

_init_children_pdv_args(*ini_pdv*, *act_args*)

get package names of the portions specified as command line args, optionally filtered by –branch option.

Return type
List[Dict[str, Any]]

_init_children_presets(*chi_vars*)

Return type
Dict[str, Set[str]]

_patch_outsourced(*file_name*, *content*, *patcher*)

Return type
str

_pp(*output*)

Return type
str

_print_pdv(*pdv*)

_register_template(*import_name*, *dev_require*, *add_req*, *tpl_projects*)

Return type
Dict[str, str]

_renew_prj_dir(*new_pdv*)

_renew_project(*ini_pdv*, *project_type*)

Return type
Dict[str, Any]

_renew_local_root_req_file(*pdv*)

_required_package(*import_or_package_name*, *packages_versions*)

Return type
bool

_template_projects(*pdv*)

determine template projects of namespace, project type and generic project (the highest priority first).

Return type
List[Dict[str, str]]

_template_version_option(*import_name*)

Return type
str

_wait()

_write_commit_message(*pdv*, *pkg_version='{project_version}'*, *title=''*)

class RemoteHost

Bases: object

base class registering subclasses as remote host repo class in *REGISTERED_HOSTS_CLASS_NAMES*.

name_prefix: str = 'repo'

create_branch: Callable

release_project: Callable

repo_obj: Callable

request_merge: Callable

classmethod __init_subclass__(**kwargs)

register remote host class name; called on declaration of a subclass of *RemoteHost*.

_repo_merge_src_dst_fork_branch(*ini_pdv*)

Return type
Tuple[Union[Repository, Project], Union[Repository, Project], bool, str]

_release_project(*ini_pdv*, *version_tag*)

class GithubCom

Bases: *RemoteHost*

remote connection and actions on remote repo in gitHub.com.

connection: Github

connection to GitHub host

connect(*ini_pdv*)

connect to gitHub.com remote host.

Parameters
ini_pdv (Dict[str, Any]) – project dev vars (host_token).

Return type
bool

Returns
True on successful authentication else False.

create_branch(*group_repo*, *branch_name*, *tag_name*)

create new remote branch onto/from tag name.

Parameters

- **group_repo** (str) – string with owner-user-name/repo-name of the repository, e.g. "UserName/RepositoryName".
- **branch_name** (str) – name of the branch to create.
- **tag_name** (str) – name of the tag/ref to create branch from.

init_new_repo(*group_repo*, *project_desc*)

config new project repo.

Parameters

- **group_repo** (str) – project owner user and repository names in the format "username/repo-name".
- **project_desc** (str) – project description.

repo_obj(*err_code*, *err_msg*, *group_repo*)

convert user repo names to a repository instance of the remote api.

Parameters

- **err_code** (int) – error code, pass 0 to not quit if project not found.
- **err_msg** (str) – error message to display on error with optional {name} to be automatically substituted with the project name from the *group_repo_names* argument.
- **group_repo** (str) – string with owner-user-name/repo-name of the repository, e.g. "UserName/RepositoryName".

Return type
Repository

Returns
python-github repository if found, else return None if err_code is zero else quit.

static _protect_branches(*project_repo*, *branch_masks*)

fork_project(*ini_pdv*, *forked_usr_repo*)

create/renew fork of a remote repo specified via the 1st argument, into our user/group namespace.

push_project(*ini_pdv*, *branch_name=''*)

push current/specified branch of project/package to remote host domain, version-tagged if release is True.

Parameters

- **ini_pdv** (Dict[str, Any]) – project dev vars.
- **branch_name** (str) – optional branch name to push (alternatively specified by the branch command line option).

release_project(*ini_pdv*, *version_tag*)

update local MAIN_BRANCH from origin and if pip_name is set then also release the latest/specified version.

Parameters

- **ini_pdv** (Dict[str, Any]) – project dev vars.
- **version_tag** (str) – push version tag in the format v<version-number> to release or LATEST to use the version tag of the latest git repository version.

request_merge(*ini_pdv*)

request merge of the origin=fork repository into the main branch at remote/upstream=forked.

class GitlabCom

Bases: *RemoteHost*

remote connection and actions on gitlab.com.

connection: Gitlab

connection to Gitlab host

connect(*ini_pdv*)

connect to gitlab.com remote host.

Parameters
ini_pdv (Dict[str, Any]) – project dev vars (REPO_HOST_PROTOCOL, host_domain, host_token).

Return type
bool

Returns
True on successful authentication else False.

create_branch(*group_repo*, *branch_name*, *tag_name*)

create new remote branch onto/from tag name.

Parameters

- **group_repo** (str) – string with owner-user-name/repo-name of the repository, e.g. "UserName/RepositoryName".

- **branch_name** (str) – name of the branch to create.
- **tag_name** (str) – name of the tag/ref to create branch from.

init_new_repo(*ini_pdv*)

create group/user project specified in ini_pdv or quit with error if group/user not found.

Parameters
ini_pdv (Dict[str, Any]) – project dev vars.

repo_obj(*err_code*, *err_msg*, *group_repo*)

convert group/project_name or an endswith-fragment of it to a Project instance of the remote repo api.

Parameters

- **err_code** (int) – error code, pass 0 to not quit if project not found.
- **err_msg** (str) – error message to display on error with optional {name} to be automatically substituted with the project name from the *group_repo* argument.
- **group_repo** (str) – group/project-name to search for.

Return type
Project

Returns
python-gitlab project instance if found, else return None if err_code is zero else quit.

project_owner(*ini_pdv*)

determine owner (group|user) of the project specified by ini_pdv or quit with error if group/user not found.

Parameters
ini_pdv (Dict[str, Any]) – project dev vars.

Return type
Union[Group, User]

Returns
instance of Group or User, determined via the user-/group-names specified by ini_pdv.

clean_releases(*ini_pdv*)

delete local+remote release tags and branches of the specified project that got not published to PYPI.

Return type
List[str]

fork_children(*ini_pdv*, **children_pdv*)

fork children of a namespace root project or of a parent folder.

fork_project(*ini_pdv*)

create/renew fork of a remote repo specified via the `package` option, into our user/group namespace.

push_children(*ini_pdv*, **children_pdv*)

push specified children projects to remotes/origin.

push_project(*ini_pdv*, *branch_name=''*)

push current/specified branch of project/package to remote host domain, version-tagged if release is True.

Parameters

- **ini_pdv** (Dict[str, Any]) – project dev vars.
- **branch_name** (str) – optional branch name to push (alternatively specified by the `branch` command line option).

release_children(*ini_pdv*, **children_pdv*)

release the latest versions of the specified parent/root children projects to remotes/origin.

release_project(*ini_pdv*, *version_tag*)

update local MAIN_BRANCH from origin and if pip_name is set then also release the latest/specified version.

Parameters

- **ini_pdv**¶ (Dict[str, Any]) – project dev vars.
- **version_tag**¶ (str) – push version tag in the format v<version-number> to release or LATEST to use the version tag of the latest git repository version.

request_children_merge(*ini_pdv*, **children_pdv*)

request the merge of the specified children of a parent/namespace on remote/upstream.

request_merge(*ini_pdv*)

request merge of the origin=fork repository into the main branch at remote/upstream=forked.

search_repos(*ini_pdv*, *fragment=''*)

search remote repositories via a text fragment in its project name/description.

show_children_repos(*ini_pdv*, **children_pdv*)

display remote properties of parent/root children repos.

show_repo(*ini_pdv*)

display properties of remote repository.

class PythonanywhereCom

Bases: *RemoteHost*

remote actions on remote web host pythonanywhere.com (to be specified by –domain option).

connection: *PythonanywhereApi*

requests http connection

name_prefix: str = 'web'

config variable name prefix

connect(*ini_pdv*)

connect to www. and eu.pythonanywhere.com web host.

Parameters
ini_pdv¶ (Dict[str, Any]) – parent/root project dev vars.

Return type
bool

Returns
True on successful authentication else False.

deploy_flags = {'ALL': False, 'CLEANUP': False, 'LEAN': False, 'MASKS': []}

optional flag names and default values for the actions *check_deploy()* and *deploy_project()*

deploy_differences(*ini_pdv*, *action*, *version_tag*, ***optional_flags*)

determine differences between the specified repository and web host/server (deployable and deletable files).

Parameters

- **ini_pdv**¶ (Dict[str, Any]) – project dev vars.

- **action**¶ (str) – pass 'check' to only check the differences between the specified repository and the web server/host, or 'deploy' to prepare the deployment of these differences.
- **version_tag**¶ (str) – project package version to deploy. pass LATEST to use the version tag of the latest repository version (PyPI release), or WORKTREE to deploy the actual working tree package version (including unstaged/untracked files).
- **optional_flags**¶ – optional command line arguments, documented in detail in the declaration of the action method parameter *check_deploy.optional_flags*.

Return type
tuple[str, str, set[str], set[str]]

Returns
tuple of 2 strings and 2 sets. the first string contains a description of the project and the server to check/deploy-to, and the second the path to the project root folder. the two sets containing project file paths, relative to the local/temporary project root folder, the first one with the deployable files, and the 2nd one with the removable files.

check_deploy(*ini_pdv*, *version_tag*, ***optional_flags*)

check all project package files at the app/web server against the specified package version.

Parameters

- **ini_pdv**¶ (Dict[str, Any]) – project dev vars.
- **version_tag**¶ (str) – version tag in the format v<version-number> to check or LATEST to check against the latest repository version or WORKTREE to check directly against the local work tree (with the locally added, unstaged and changed files).
- **optional_flags**¶ – additional/optionally supported command line arguments:
  - ALL is including all deployable package files, instead of only the new, changed or deleted files in the specified repository.
  - CLEANUP is checking for deletable files on the web server/host, e.g. after they got removed from the specified repository or work tree.
  - LEAN is reducing the deployable files sets to the minimum (using e.g. the function *skip_files_lean_web()*), like e.g. the gettext .po files, the media_ini root folder, and the static sub-folder with the initial static files of the web project.
  - MASKS specifies a list of file paths masks/pattern to be included in the repository files to check/deploy. to include e.g. the files of the static root folder specify this argument as MASKS="['static/**/*']". single files can be included too, by adding their possible file names to the list - only the found ones will be included. for example to include the django database you could add some possible DB file names to the list like in "MASKS=['static/**/*', 'db.sqlite', 'project.db']"

deploy_project(*ini_pdv*, *version_tag*, ***optional_flags*)

deploy code files of django/app project version to the web-/app-server.

Parameters

- **ini_pdv**¶ (Dict[str, Any]) – project dev vars.
- **version_tag**¶ (str) – version tag in the format v<version-number> to deploy or LATEST to use the tag of the latest repository version or WORKTREE to deploy directly from the local work tree (including locally added, unstaged and changed files).
- **optional_flags**¶ – optional command line arguments, documented in the *check_deploy()* action.

add_children_file(*ini_pdv*, *file_name*, *rel_path*, **children_pdv*)

add a file, template of outsourced text file to the working trees of parent/root and children/portions.

Parameters

- **ini_pdv** (Dict[str, Any]) – parent/root project dev vars.
- **file_name** (str) – source (template) file name (optional with path).
- **rel_path** (str) – relative destination path within the working tree.
- **children_pdv** (Dict[str, Any]) – project dev vars of the children to process.

Return type
bool

Returns
True if the file got added to the parent/root and to all children, else False.

add_file(*ini_pdv*, *file_name*, *rel_path*)

add file, template or outsourced text file into the project working tree.

Parameters

- **ini_pdv** (Dict[str, Any]) – project dev vars.
- **file_name** (str) – file name to add (optional with abs. path, else relative to working tree root folder).
- **rel_path** (str) – relative path in destination project working tree.

Return type
bool

Returns
True if the file got added to the specified project, else False.

bump_version(*ini_pdv*)

increment project version.

check_children_integrity(*parent_pdv*, **children_pdv*)

run integrity checks for the specified children of a parent or portions of a namespace.

check_integrity(*ini_pdv*)

integrity check of files/folders completeness, outsourced/template files update-state and CI tests.

clone_children(*parent_or_root_pdv*, **project_versions*)

clone specified namespace-portion/parent-child repos to the local machine.

Parameters

- **parent_or_root_pdv** (Dict[str, Any]) – vars of the parent/namespace-root project.
- **project_versions** (str) – package/project names with optional version of the children to be cloned.

Return type
List[str]

Returns
list of project paths of the cloned children projects (for unit testing).

clone_project(*ini_pdv*, *package_or_portion=''*)

clone remote repo to the local machine.

Parameters

- **ini_pdv** (Dict[str, Any]) – vars of the project to clone.
- **package_or_portion** (str) – name of the package/portion to clone, optionally with version number.

Return type
str

Returns
project path of the cloned project (used for unit tests).

commit_children(*ini_pdv*, **children_pdv*)

commit changes to children of a namespace/parent using the individually prepared commit message files.

commit_project(*ini_pdv*)

commit changes of a single project to the local repo using the prepared commit message file.

delete_children_file(*ini_pdv*, *file_name*, **children_pdv*)

delete file or empty folder from parent/root and children/portions working trees.

Parameters

- **ini_pdv** (Dict[str, Any]) – parent/root project dev vars.
- **file_name** (str) – file/folder name to delete (optional with path, relative to working tree root folder).
- **children_pdv** (Dict[str, Any]) – tuple of children project dev vars.

Return type
bool

Returns
True if file got found & deleted from the parent and all children projects, else False.

delete_file(*ini_pdv*, *file_name*)

delete file or empty folder from project working tree.

Parameters

- **ini_pdv** (Dict[str, Any]) – project dev vars.
- **file_name** (str) – file/folder name to delete (optional with path, relative to working tree root folder).

Return type
bool

Returns
True if the file got found and delete from the specified project, else False.

install_children_editable(*ini_pdv*, **children_pdv*)

install parent children or namespace portions as editable on local machine.

install_editable(*ini_pdv*)

install project as editable from source/project root folder.

new_app(*ini_pdv*)

create or complete/renew a gui app project.

Return type
Dict[str, Any]

new_children(*ini_pdv*, **children_pdv*)

initialize or renew parent folder children or namespace portions.

Return type
List[Dict[str, Any]]

new_django(*ini_pdv*)

create or complete/renew a django project.

Return type
Dict[str, Any]

new_module(*ini_pdv*)

create or complete/renew module project.

Return type
Dict[str, Any]

new_namespace_root(*ini_pdv*)

create or complete/renew namespace root package.

Return type
Dict[str, Any]

new_package(*ini_pdv*)

create or complete/renew package project.

Return type
Dict[str, Any]

new_project(*ini_pdv*)

complete/renew an existing project.

Return type
Dict[str, Any]

prepare_children_commit(*ini_pdv*, *title*, **children_pdv*)

run code checks and prepare/overwrite the commit message file for a bulk-commit of children projects.

Parameters

- **ini_pdv** (Dict[str, Any]) – parent/root project dev vars.
- **title** (str) – optional commit message title.
- **children_pdv** (Dict[str, Any]) – tuple of project dev vars of the children to process.

prepare_commit(*ini_pdv*, *title=''*)

run code checks and prepare/overwrite the commit message file for the commit of a single project/package.

Parameters

- **ini_pdv** (Dict[str, Any]) – project dev vars.
- **title** (str) – optional commit message title.

refresh_children_outsourced(*ini_pdv*, **children_pdv*)

refresh outsourced files from templates in namespace/project-parent children projects.

refresh_outsourced(*ini_pdv*)

refresh/renew all the outsourced files in the specified project.

rename_children_file(*ini_pdv*, *old_file_name*, *new_file_name*, **children_pdv*)

rename file or folder in parent/root and children/portions working trees.

Parameters

- **ini_pdv** (Dict[str, Any]) – parent/root project dev vars.
- **old_file_name** (str) – file/folder name to rename (optional with path, relative to working tree root folder).
- **new_file_name** (str) – new name of file/folder (optional with path, relative to working tree root folder).
- **children_pdv** (Dict[str, Any]) – tuple of project dev vars of the children to process.

Return type
bool

Returns
True if the file got renamed in the parent and all children projects, else False.

rename_file(*ini_pdv*, *old_file_name*, *new_file_name*)

rename file or folder in project working tree.

Parameters

- **ini_pdv** (Dict[str, Any]) – project dev vars.
- **old_file_name** (str) – source file/folder (optional with path, absolute or relative to project working tree).
- **new_file_name** (str) – destination file/folder (optional path, absolute or relative to project working tree).

Return type
bool

Returns
True if file/folder got renamed, else False.

run_children_command(*ini_pdv*, *command*, **children_pdv*)

run console command for the specified portions/children of a namespace/parent.

Parameters

- **ini_pdv** (Dict[str, Any]) – parent/root project dev vars.
- **command** (str) – console command string (including all command arguments).
- **children_pdv** (Dict[str, Any]) – tuple of children project dev vars.

show_actions(*ini_pdv*)

get info of available/registered/implemented actions of the specified/current project and remote.

show_children_status(*ini_pdv*, **children_pdv*)

run integrity checks for the specified portions/children of a namespace/parent.

show_children_versions(*ini_pdv*, **children_pdv*)

show package versions (local, remote and on pypi) for the specified children of a namespace/parent.

show_status(*ini_pdv*)

show git status of the specified/current project and remote.

show_versions(*ini_pdv*)

display package versions of worktree, remote/origin repo, latest PyPI release and default app/web host.

update_children(*ini_pdv*, **children_pdv*)

fetch and rebase the MAIN_BRANCH to the local children repos of the parent/namespace-root(also updated).

update_project(*ini_pdv*)

fetch and rebase the MAIN_BRANCH of the specified project in the local repo.

Return type
List[str]

prepare_and_run_main()

prepare and run app

main()

main app script

3.7 aedev.pythonanywhere

web api for www.pyanywhere.com and eu.pyanywhere.com

a similar package can be found at **`https://gitlab.com/texperience/pythonanywhereapiclient`__**

Classes

PythonanywhereApi(web_domain, web_user, ...)	remote host api to a project package on the web hosts eu.pythonanywhere.com and pythonanywhere.com.

class PythonanywhereApi(*web_domain*, *web_user*, *web_token*, *project_name*)

Bases: object

remote host api to a project package on the web hosts eu.pythonanywhere.com and pythonanywhere.com.

__init__(*web_domain*, *web_user*, *web_token*, *project_name*)

initialize web host api and the deployed project package name.

Parameters

- **web_domain** (str) – remote web host domain.
- **web_user** (str) – remote connection username.
- **web_token** (str) – personal user credential token string on remote host.
- **project_name** (str) – name of the web project package.

property error_message: str

error message string if an error occurred or an empty string if not.

Getter
return the accumulated error message of the recently occurred error(s).

Setter
any assigned error message will be accumulated to recent error messages. pass an empty string to reset the error message.

property project_name: str

project main package name string property.

Getter
return the currently connected/configured project package name of the web host server.

Setter
set/change the currently connected/configured project name of the web host server.

_folder_items(*folder_path*)

Return type
Optional[list[dict[str, str]]]

_from_json(*response*)

convert json in response to python type (list/dict).

Parameters
response (Response) – response from requests to convert into python data type.

Return type
Union[list[dict[str, Any]], dict[str, dict[str, Any]], None]

Returns
[list of] dictionaries converted from the response content or None on error.

_prepare_collector(*skipper*)

_request(*url_path*, *task*, *method=<function get>*, *success_codes=(200, 201)*, ***request_kwargs*)

send a https request specified via *method* and return the response.

Parameters

- **url_path** (str) – sub url path to send request to.
- **task** (str) – string describing the task to archive (used to compile an error message).
- **method** (Callable) – requests method (get, post, push, delete, patch, ...).
- **success_codes** (Sequence) – sequence of response.status_code success codes
- **request_kwargs** – additional request method arguments.

Return type
Response

Returns
request response. if on error occurred then the instance string attribute *error_message* contains an error message. if the caller is not checking for errors and not resetting the error message string, then this function will accumulate further errors to *error_message*, separated by two new line characters.

available_consoles()

determine the available consoles.

Return type
Optional[list[dict[str, Any]]]

Returns
python dictionary with available consoles or None if an error occurred.

deployed_code_files(*path_masks*, *skip_file_path=<function PythonanywhereApi.<lambda>>*)

determine all deployed code files of given package name deployed to the pythonanywhere server.

Parameters

- **path_masks** (Sequence[str]) – root package paths with glob wildcards to collect deployed code files from.
- **skip_file_path** (Callable[[str], bool]) – called for each found file/folder with the path_mask relative to the package root folder as argument, returning True to exclude the specified item from the returned result set. calls of a folder have a prefix of a slash character followed by a dot (*"/."*) and help to minimize the number of calls against the web server api.

Return type
Optional[set[str]]

Returns
set of file paths of the package deployed on the web, relative to the project root or None if an error occurred.

deployed_file_content(*file_path*)

determine the file content of a file deployed to a web server.

Parameters
file_path (str) – path of a deployed file relative to the project root.

Return type
Optional[bytes]

Returns
file content as bytes or None if error occurred (check self.error_message).

deployed_version()

determine the version of a deployed django project package.

Return type
str

Returns
version string of the package deployed to the web host/server or empty string if package version file or version-in-file not found.

deploy_file(*file_path*, *file_content*)

add or update a project file to the web server.

Parameters

- **file_path** (str) – path relative to the project root of the file to be deployed (added or updated).
- **file_content** (bytes) – file content to deploy/upload.

Return type
str

Returns
error message if update/add failed else on success an empty string.

`delete_file_or_folder`(*file_path*)

delete a file or folder on the web server.

Parameters
file_path¶ (str) – path relative to the project root of the file to be deleted.

Return type
str

Returns
error message if deletion failed else on success an empty string.

`files_iterator`(*path_mask*, *level_index=0*)

find files matching the path mask string passed as the paramref:.*path_mask* argument.

Parameters

- **path_mask**¶ (str) – file path pattern/mask with optional wildcards. passing an empty string will return the files of the project/package root directory, as well as passing '.' or '*'. also absolute path masks will be relative to the project root directory. file path mask matches are case-sensitive (done with the function `fnmatch.fnmatchcase()`).
- **level_index**¶ (int) – folder level depth in *passed file path mask* to start searching (only specified in recursive call).

Return type
Iterable[Any]

Returns
iterator/generator yielding dicts. each dict has a *file_path* key containing the path string of the found file relative to the project root folder and a *type* key containing the string *'directory'* or `'file'

`find_project_files`(*path_mask=''*, *skip_file_path=<function PythonanywhereApi.<lambda>>*, *collector=None*)

determine the files matching the glob pattern provided in *path_mask* at the app/web server.

not using the files tree api endpoints/function (f"files/tree/?path=/home/{self.web_user}/{project_name}") because their response is limited to 1000 files (see https://help.pythonanywhere.com/pages/API#apiv0userusernamefilestreepathpath) and e.g. kairos has more than 5300 files in its package folder (mainly for django filer and the static files).

Parameters

- **path_mask**¶ (str) – file mask including relative path to the package project root to be searched. passing an empty string (the default) returns all files in the package root directory.
- **collector**¶ (Optional[Collector]) – file collector callable.
- **skip_file_path**¶ (Callable[[str], bool]) – called for each found file/folder with the path_mask relative to the package root folder as argument, returning True to exclude the specified item from the returned result set. calls of a folder have a prefix of a slash character followed by a dot (*"/."*) and help to minimize the number of calls against the web server api.

Return type
Optional[set[str]]

Returns
set of file paths of the package deployed on the web, relative to the project root or None if an error occurred. all files underneath a

3.8 aedev.setup_hook

3.8.1 individually configurable setup hook

by replacing the function call of *project_env_vars()* in the *aedev.setup_project* module of a project with a call to the function *hooked_project_env_vars()*, provided by this module, an individual hook will be executed automatically just before the execution of the `setuptools` `setup()` method.

this setup hook can be used e.g. to adapt/extend the variables and constants of the project environment variable mapping. also, the settings of a portion of a namespace project can be configured individually by adding a hook module (containing a hook method) into the portions project working tree root folder. the default names of the hook module and method are specified by the constant *NAMESPACE_EXTEND_ENTRY_POINT*, which gets also added automatically into the project environment variable mapping by the function *hooked_project_env_vars()* of this module.

Module Attributes

NAMESPACE_EXTEND_ENTRY_POINT	module:method default names of the setup hook

Functions

hooked_project_env_vars([project_path])	called from setup.py instead of *project_env_vars()* to determine the project vars.
pev_update_hook(pev)	check if optional NAMESPACE_EXTEND_ENTRY_POINT hook file exists and if yes then run it to change pev values.

NAMESPACE_EXTEND_ENTRY_POINT = 'setup_hooks:extend_project_env_vars'

module:method default names of the setup hook

hooked_project_env_vars(*project_path=''*)

called from setup.py instead of *project_env_vars()* to determine the project vars.

Parameters
project_path (str) – optional rel/abs path of package/app/project root (def=current working directory).

Return type
Dict[str, Union[str, Sequence[str], List[Tuple[str, Tuple[str, ...]]], Dict[str, Any]]]

Returns
project environment variables mapping/dict, optionally updated by hook.

pev_update_hook(*pev*)

check if optional NAMESPACE_EXTEND_ENTRY_POINT hook file exists and if yes then run it to change pev values.

Parameters

pev (Dict[str, Union[str, Sequence[str], List[Tuple[str, Tuple[str, ...]]], Dict[str, Any]]]) – namespace environment variables.

Return type

bool

Returns

True if hook got called, else False.

CHAPTER

FOUR

MANUALS AND TUTORIALS

4.1 git repository manager user manual

4.1.1 installation of grm

to installing this tool open a console window and run the following command:

```
.. code-block:: shell

   pip install aedev_git_repo_manager
```

after the installation the `grm` command will be available in your OS console.

4.1.2 usage of grm

`grm` is supporting you on all devops (development operations) of your python library, application and web projects.

this covers all actions done on your local machine, on your repository host servers (like `gitlab.com` or `github.com`) and on your web and applikcation deployment servers, like:

- creating new projects
- maintaining and upgrading existing projects
- running integrity checks and unit tests
- maintaining, syncing and pushing of your git repositories
- creating and maintaining merge requests on your git repository servers
- release of your project onto the cheese shop (PyPI.com)
- deployment of your app/web project

command line options and action arguments

the git repository manager command line consist of options, action keywords, action arguments and optional action argument flags:

```
.. code-block:: shell

   grm [options] [action-keywords] [action-arguments] [action-flags]
```

all command line options are available in a long form, preceded with two leading hyphen characters, and as a single character in a short form, preceded with a single hyphen character.

executing `grm` with the *–help* command line option (short *-h*) displays a short summary of the available command line options:

```
.. code-block:: shell

   grm --help
```

general command line options like e.g. *–verbose* (*-v*), *–debug_level* (*-D*), *–path* (*-p*) or *–project* (*-P*) can be specified for any action. other options, like e.g. the filter options *–filterBranch* (*-B*) and *–filterExpression* (*-F*), are only supported for bulk actions.

execute `grm` with the *show_actions()* action to display a brief summary of all the available/registered actions for a project:

```
.. code-block:: shell

   grm show_actions
```

the `action-keywords` argument is composed of several words, seperated by either a space character, a hyphen character or an underscore character. some actions can even be abbreviated by a single word shortcut. therefore the following four commands are identical/equivalent:

```
.. code-block:: shell

   grm show_actions
   grm show-actions
   grm show actions
   grm actions
```

you can add the *–verbose* and/or –debug_level command line options to get a more verbose output. e.g. to include for each listed action also their `action-arguments` and `action-flags`, their supported project types and their action shortcut, simply add these options to command line of the *show_actions()* action:

```
.. code-block:: shell

   grm --verbose --debug_level=2 show_actions
```

the equivalent command line using the short option form (with only one leading hyphen character), and the shortcut of the *show_actions()* action (which is *actions*) would look like:

```
.. code-block:: shell

   grm -v -D 2 actions
```

some actions are expecting additional `action-arguments`.

e.g. to execute the *release_project()* action the *aedev.git_repo_manager.__main__.GitlabCom.release_project.version_tag* action argument has to specify the project version to release.

for some actions you can optionally specify `action-flags`. each flag has a default value, which will be used if the flag is not specified on the command line.

the action *check_deploy()* e.g. is supporting the flag *CLEANUP* with a default value of `False`. specifying this flag on the command line is switching the flag value to `True`. the resulting flag value can also be specified on the command line by adding an equal character ('=') to the flag name, directly followed by the flag value. so the following two commands are identical:

```
.. code-block:: shell

   grm check_deploy ... CLEANUP
   grm check_deploy ... CLEANUP=True
```

bulk actions

most of the `grm` actions operate on a single project or repository and should be executed in the root folder of the project working tree.

some of them are also available as bulk actions, which are affecting multiple projects, e.g. the portions of a namespace, or the projects located under the same parent directory.

bulk actions on portions of a namespace are processed by executing them in the namespace root project root folder.

bulk actions on projects underneath a parent directory are executed in the parent folder.

alternatively they can be executed from any other folder by specifying the namespace root project or the projects parent folder via the *–project* or *–path* options.

for example, bulk actions on namespace root project, like e.g. the ae namespace root project via `--project ae_ae` or the the aedev namespace root project via `--path path/to/aedev_aedev`.

Hint: bulk actions are recognizable by the additional `children` keyword in their action name.

---

repository status actions

several actions are determining the project(s) status, like e.g. *show_status()*, *show_children_status()*, *show_repo()*, *show_children_repos()*, *check_integrity()*, *check_children_integrity()*, *show_versions()*, *show_children_versions()*, and *search_repos()*.

project and repository maintenance actions

useful actions to create, extend or renew a project repository respectively multiple project repositories, are e.g. *new_app()*, *new_children()*, *new_django()*, *new_module()*, *new_namespace_root()*, *new_package()*, *new_project()*, *bump_version()*, *refresh_outsourced()*, *refresh_children_outsourced()*.

actions for your other repository maintenance workflows are e.g. *clone_project()*, clone_children_project(), *fork_project()*, *fork_children()*, *prepare_commit()*, *prepare_children_commit()*, *commit_project()*, *commit_children()*, *push_project()*, *push_children()*, *request_merge()*, *request_children_merge()*, *release_project()*, *release_children()*, *install_editable()*, and *install_children_editable()*.

to manipulate single files in project repositories use the actions *add_file()*, *add_children_file()*, *delete_file()*, *delete_children_file()*, *rename_file()*, *rename_children_file()*.

in order to synchronize the local *MAIN_BRANCH* branch with any changes done to same branch on the 'origin' remote, execute grm with the *update_project()* and *update_children()* actions.

the *clean_releases()* action deletes local+remote release tags and branches of the specified project that got not published to PYPI.

the execution of bulk command lines for a group of projects can be done with the *run_children_command()* action.

web and app deployment server actions

the actions *check_deploy()* and *deploy_project()* are working directly with the deployment servers of your web/Django or app project.

check_deploy() is comparing the deployed files against any repository version tag or against the package files in your project work tree.

the *deploy_project()* action is deploying any new or changed package files to your web or app deployment server.

filtering children of projects parent or portions of a namespace

which children will get processed in a bulk actions get specified by a children-set-expression action argument, which can combine one or more of the following placeholders via the set operators of python (| for union, *&* for intersection, - for difference and ^ for symmetric difference):

- *all*: all children projects
- *editable*: projects installed as editable
- *modified*: projects having uncommitted changes
- *develop*: projects having checked-out the *MAIN_BRANCH*
- *filterBranch*: projects having checked-out the branch specified with the *–filterBranch* (short *-B*) option
- *filterExpression*: projects matching the expression specified with the *–filterExpression* (short *-F*) option

for example to show the project versions of namespace portions with uncommitted changes, execute the following command in the root folder of the namespace root project:

```
.. code-block:: shell

   grm show_children_versions modified
```

to additionally restrict the last example to projects with uncommitted changes in the *MAIN_BRANCH* run:

```
.. code-block:: shell

   grm show_children_versions "modified & develop"
```

Note: children-set-expression with set-operators have to be included into high-commas.

more flexible filtering can be done with the command line options *–filterExpression* and *–filterBranch*. by specifying one of these options the selected/filtered children are then available as a children-set-expression with the same name as the specified option.

for example to only show the versions of projects with uncommitted changes in the branch `branch_name` run:

```
.. code-block:: shell

   grm --filterBranch=branch_name show_children_versions "modified & filterBranch"
```

Note: the name of the branch get specified with the *–filterBranch* option. and the name of the option can then be used like a python set in the children-set-expression action argument.

in general, any bulk action can be restricted to only process children/portions projects that have the specified branch name checked-out. e.g. to only process all children that have checked out the branch `branch_name` run:

```
.. code-block:: shell

   grm --filterBranch=branch_name <any_bulk_action> filterBranch
```

exactly the same selection result could be achieved via a more complex Python expression, using the *–filterExpression* option/children-set-expression:

```
.. code-block:: shell

   grm --filterExpression="_git_current_branch(chi_pdv)=='branch_name'" <any_bulk_
↪action> filterExpression
```

Hint: the filter expression can contain project environment variables and any globals of the git-repo-manager tool. additionally, the variable *chi_pdv* can be used to additionally access the project environment variables of the other children/portion projects.

Note: filter expressions should be included in high-commas.

the next example is selection all children with a project package version number below or equal to `0.2`:

```
.. code-block:: shell

   grm --filterExpression "project_version<='0.2'" <children_bulk_action>␣
↪filterExpression
```

the example underneath is showing the local, remote and PyPI versions of the children projects that have a branch (checked-out or not) with the name `branch_name` in their repository:

```
.. code-block:: shell

    grm -F "'branch_name' in _git_branches(chi_pdv)" show_children_versions␣
↪filterExpression
```

to bulk-release multiple children projects in a *contribution process* workflow, the following bulk actions can be executed, e.g. from within the root folder of a namespace root project:

- *new_children()* to increment the versions and refresh outsourced files from templates:

```
.. code-block:: shell

    grm -b=branch_name new_children modified
```

- *prepare_children_commit()* to prepare the commit message files (after you implemented all changes into the above created branch with the name `branch_name`):

```
.. code-block:: shell

    grm prepare_children_commit "commit message for branch_name" modified
```

- *commit_children()* to commit changes to the local git repositories:

```
.. code-block:: shell

    grm commit_children modified
```

- *push_children()* to push the committed changes to the remote repositories:

```
.. code-block:: shell

    grm --filterBranch=branch_name push_children filterBranch
```

- *request_children_merge()* to merge pushed changes to the main branches on the remote host (without a repository forg add the options: -f -u=group_or_user_name):

```
.. code-block:: shell

    grm --filterBranch=branch_name request_children_merge filterBranch
```

- *release_children()* to bulk-release the project packages to PyPI:

```
.. code-block:: shell

    grm --filterBranch=branch_name release_children filterBranch
```

- *install_children_editable()*: updates/updates editable installations of your local projects into your virtual environment:

```
.. code-block:: shell

    grm -F "'branch_name' in _git_branches(chi_pdv)" install_children_editable␣
↪filterExpression
```

remote server configuration

grm supports two types of remote servers. remote servers that are hosting the project repository (e.g. on gitlab.com or github.com), and remote servers that are hosting deployable apps or web sites (e.g. pythonanywhere.com).

grm actions with write access to any remote host (web/repository server), like e.g. *deploy_project()* or *push_project()*, are requesting the user credentials for the authentication from the remote server configurations.

Note: remote server configurations can be specified in multiple ways. configuration options specified to grm via command line arguments have the highest priority, followed by OS environment variables, *grm config variables*, and the *remote server configuration* files.

---

command line config options

the remote server domain address can be specified via command line config-option *domain*.

user credentials can be specified via the grm command line config options: *token* and *user* or *group*.

grm config variables

user credentials not specified by the command line config options are determined from the application config variable via a user- and domain-specific lookup with the help of the get_variable() method.

for example to resolve the value of the not specified *token* command line option, the lookup first checks if there exists an OS environment variable (also via the python-dotenv package), and if not found then it is looking for an application config variable.

the lookup of the value of the *token* option, for an user with the name michael at the domain www.example.com, is done in the following order:

- OS environment variable AE_OPTIONS_HOST_TOKEN_AT_WWW_EXAMPLE_COM_MICHAEL
- config variable host_token_at_www_example_com_michael in the config section aeOptions
- OS environment variable AE_OPTIONS_HOST_TOKEN_AT_WWW_EXAMPLE_COM
- config variable host_token_at_www_example_com in the config section aeOptions

project development variables

this type of configuration variables are extending the project environment variables provided by the *aedev.setup_project* module. the variable values can be configured via the files *pev.defaults* and/or *pev.updates*, situated in the root folder of a project working tree.

Note: the content of these two files consists of a single Python dictionary literal.

---

by providing for example the user name (of your account at your repository host server) in the key *STK_AUTHOR*, you don't need to specify it any longer on the grm command line via the –user option. in the most cases you will want to provide also the (user/group) name of the repository owner in the key *REPO_GROUP*, which results in the file contents:

```
{
    'STK_AUTHOR': 'UserName',
    'REPO_GROUP': 'UserOrGroupName',
}
```

Hint: the default value of *REPO_GROUP* gets compiled from the project name followed by a hyphen and the word *group*.

in order to change the default domain (`gitlab.com`) of the git repository server/host for a project to `github.com`, add also the following two lines have into this dict literal:

```
'REPO_CODE_DOMAIN': 'github.com',
'REPO_PAGES_DOMAIN': 'github.io',
```

git credential storage

the git credential storage can be used as the last fallback if your user credentials are neither specified as *command line config options* nor via the *grm config variables*.

the user credentials for actions on git repository hosts (`gitlab.com/github.com/...`) like *push_project()* can alternatively be set and stored via the git configuration settings (see https://git-scm.com/book/en/v2/Git-Tools-Credential-Storage and https://stackoverflow.com/questions/46645843).

Hint: see **`https://stackoverflow.com/questions/65163081`__** to disable user/password prompts for fetch and check actions to git repository hoster that don't need authentication (and not using the *token* option), like e.g. *_git_fetch()*.

4.1.3 grm example workflows

this section describes some typical development workflows managed with the help of the `grm` tool.

create a new project

to create a new project (in this example a small console app module with the project name `senv`), first create a folder with the name of the new project directly underneath of your projects source parent folder (e.g. `~/src`).

then, within the new project root folder create the file `pev.defaults` (or `pev.updated`, like explained in the section *project development variables*), in order to specify your user name, your credentials, or the repository owner at your repository host server.

Hint: the section *remote server configuration* describes how you can configure default values of your repository server, your user account name and credentials, if you not want to specify them in every run of `grm` via the command line options.

optionally create and activate a virtual environment for the new project, like e.g. `aedev39` with the tool pyenv:

```
pyenv local aedev39
```

Hint: using `pyenv local` has the advantage to ensure that the project's virtual environment gets activated automatically, as soon as you change the current directory of your console to the project root folder.

make sure you have installed the `grm` tool, by running the following command within your new project root folder:

```
pip install aedev-git-repo-manager
```

now you can run the `grm` tool for the first time in order to create the initial git repository, and some basic files for the specified project type (e.g. a module project):

```
grm new_module
```

the grm action `new_module` specifies the type of the project as a single module. for a more complex project (including multiple modules) use instead the `new_package` action, for a namespace root project the `new_namespace_root` action, for a GUI application the `new_app` action, and for a django project the `new_django` action.

Hint: for a console app, in controary to a GUI app, use either the `new_module` or `new_package` actions.

now you can start completing the unit and integration tests code by editing the prepared file `tests/test_senv.py`. then the project code can be amended in the generated file `senv.py`.

Note: the project code of a project of the `package` type resides instead in the file `senv/__init__.py`, respective in `<namespace_name>/senv/__init__.py` for a namespace portion package.

to complete the documentation of your new project, amend the prepared files `README.md`, `docs/index.rst` and `docs/features_and_examples.rst` accordingly.

Hint: at any time in the implementation process you can run the two actions *renew* and *check* in order to keep the files created from templates up-to-date, and to test your implementation:

```
grm refresh
grm check
```

after finishing the implementation you can create the commit message file `.commit_msg.txt` in the project root folder with the *prepare* action:

```
grm prepare
```

the content of the commit message file can then be ammended with additional notes.

to commit the first implementation of your new project into your local git repository, execute the *commit* action:

```
grm commit
```

now the new project can be pushed to your repository host server (`gitlab.com` by default) by executing the *push* action:

```
grm push
```

change request on an existing project

a typical workflow to change or add code of an already existing project gets processed with the following `grm` actions:

- *`fork`* - create or update your fork (only for already existing porjects).
- *`renew`* - update/refresh/renew the files created from templates.
- *`prepare`* - create a commit message after all planned changes/additions are implemented.
- *`commit`* - create a new commit.
- *`push`* - push the commit to the origin repository (your fork).
- *`request`* - create a merge request.

to complete the workflow, the release and deployment of a project has to be done by an repository maintainer with the following `grm` actions:

- *`release`* - merge the changes into the main branch ({MAIN_BRANCH}) and create a new project release at PyPI.
- *`deploy`* - deployment of the new/changed project (only available for web and app projects).

setup a new Django CMS server project

the following example describes all the steps that need to be done in the bash console on your computer, in order to create a new Django project (using Django 4.2 and DjangoCms 4.1), with the project name `oaios`.

after changing your current working directory to the projects source parent folder (e.g. <username>/src), execute the following commands to create the project root folder `oaios`, and setup a new virtual environment with the name `dj4`:

```
pyenv install 3.12.0
pyenv virtualenv 3.12.0 dj4
mkdir oaios
cd oaios
pyenv local dj4
pip install --upgrade pip setuptools aedev-git-repo-manager
```

now, still from within the project root folder and with the new virtual environment activated, you can install Django 4.2 and DjangoCms 4.1 and prepare an initial project structure by executing the following commands. the `djangocms` command will prompt you to enter the name, email address and password of the django admin/superuser:

```
pip install Django==4.2
pip install django-cms (4.1.1 released on 1st of may 2024)
djangocms oaios .
```

Note: don't miss the final dot in the `djangocms` command.

now execute the following commands in order to adapt the new project to be managed by `grm` and using the template files provided by the projects in the aedev namespace

```
mv requirements.in requirements.txt
rm LICENSE
cp ../kairos/CONTRIBUTING.rst .
cp ../kairos/LICENSE.md .
```

(continues on next page)

(continued from previous page)

```
cp ../kairos/README.md .
cp ../kairos/SECURITY.md .
cp ../kairos/.gitignore .
cp ../kairos/pev.defaults .
```

Hint: alternatively to copying the template based files from another django project (`kairos` in this case), you could create them as new text files, containing the string content of the `grm` *OUTSOURCED_MARKER* in its first line.

now edit the empty project file `oaios/__init__.py`, created by the ``djangocms``command and add inthere the following content:

```
""" Our All In One Server
"""
__version__ = '0.3.0'
```

then adapt the *project development variables* specified in the file `pev.defaults` to your web project:

```
{
    'STK_AUTHOR': 'your-repo-host-account-user-name',
    'REPO_GROUP': 'your-repo-host-owner-user-or-group',     # e.g. 'oaios-group'
    'web_domain': 'your-web-app-host-domain',               # e.g. 'www.pythonanywhere.com
↪'
    'web_user':   'your-web-app-host-user-name',
}
```

Hint: to prevent the release of your web project onto the PyPI cheese shop, set the value of the *pip_name* key to an empty string.

finally, in order to:

1. complete the project files from the templates,
2. prepare the commit message,
3. commit to git repository,
4. push the commit to Gitlab,
5. create a merge request and
6. release to PyPI and reset the local project (repository) to the main branch:

run the following `grm` actions/commands:

```
grm -f -i 0 -b init_project renew
grm prepare
grm commit
grm -f push
grm -f -u ae-group request
grm release LATEST
```

Hint: the force/-f command option has to be specified in this example for the `renew` and `push` actions, in order to use the version 0.3.0 (copied from `kairos`, instead of an initial version 0.0.1), and for the `request` action in order to

create the merge/push request directly in the name of the `ae-group` maintainer(s).

additional information to setup Django/CMS

- install Django development version (finally NOT used for this project/venv): https://docs.djangoproject.com/en/4.2/topics/install/#installing-the-development-version
- install DjangoCms by hand (not using docker): https://docs.django-cms.org/en/latest/introduction/01-install.html#installing-django-cms-by-hand
- Django/Cms software versions: https://docs.djangoproject.com/en/4.2/faq/install/#faq-python-version-support and https://docs.django-cms.org/en/latest/index.html#software-version-requirements-and-release-notes

more grm example workflows

more detailed workflow examples for``aedev`` namespace portion projects can be found in the contribution documentation of a project, and for web projects in the **`programmer manual <https://kairos.readthedocs.io/en/latest/programmer_manual.html#update-from-version-vx-x-xx-to-vx-x-yy-on-pythonanywhere >`__** of the `kairos` web project.

CHAPTER

FIVE

INDICES AND TABLES

- portion repositories at gitlab.com
- genindex
- modindex
- `ae` namespace projects and documentation
- `aedev` namespace projects and documentation

PYTHON MODULE INDEX

a

INDEX

Symbols

A

B

C

D

E

R

S

T

U

V